

An underwater scene featuring a vibrant coral reef on the left side, with various types of coral in shades of green and yellow. Several striped surgeonfish, characterized by their blue bodies and black vertical stripes, are swimming throughout the scene. The background is a deep, dark blue, suggesting the open ocean. The overall composition is dynamic and visually appealing, typical of a book cover for a technical or educational work.

Hanbit eBook

Realtime 11

Thinking About

C++ STL

프로그래밍

최흥배 지음

 **한빛미디어**  
Hanbit Media, Inc.

Thinking About:  
**C++ STL 프로그래밍**

지은이\_ **최흥배**

2003년부터 지금까지 보드 게임부터 시작해서 MMORPG 게임까지 다양한 온라인 게임의 서버 프로그램을 만들어 왔다. 게임 개발자이다 보니 프로그래밍 언어 중 C++을 주력 무기로 사용하고 C#을 보조 언어로 사용하고 있다. 다른 프로그래머들과 기술이나 경험을 공유하는 것을 좋아해서 게임 개발자 커뮤니티나 세미나 강연을 통해서 교류하고 있다. 요즘은 C++11 프로그래밍과 Linux 플랫폼 프로그래밍, Mono, Node.js에 대해서 스테디하고 있다. 웹이 대중화되기 전부터 프로그래밍 공부를 해서 그런지 아직도 기술을 배울 때는 책을 더 선호하며 지금도 매달 새로운 프로그래밍 책을 보고 있다. 현재 티쓰리엔터테이먼트 삼국지천 팀에서 근무하고 있다.

● 블로그 : <http://jacking.tistory.com/>

● 트위터: @jacking75

## Thinking About: C++ STL 프로그래밍

---

초판발행 2012년 12월 21일

지은이 최흥배 / 펴낸이 김태현

펴낸곳 한빛미디어 (주) / 주소 서울시 마포구 양화로 7길 83 한빛미디어(주) IT출판부

전화 02-325-5544 / 팩스 02-336-7124

등록 1999년 6월 24일 제10-1779호

ISBN 978-89-7914-993-7 15560 / 비매품

책임편집 배용석 / 기획 스마트미디어팀 / 편집 김재룡, 박민근, 이정재, 조진희, 한상곤

디자인 표지 여동일, 내지 스튜디오 [임], 조판 스마트미디어팀

마케팅 박상용, 박주훈, 정민하

이 책에 대한 의견이나 오타자 및 잘못된 내용에 대한 수정 정보는 한빛미디어(주)의 홈페이지나 아래 이메일로 알려주십시오.

한빛미디어 홈페이지 [www.hanb.co.kr](http://www.hanb.co.kr) / 이메일 [ask@hanb.co.kr](mailto:ask@hanb.co.kr)

---

Published by HANBIT Media, Inc. Printed in Korea

Copyright © 2012 HANBIT Media, Inc.

이 책의 저작권은 한빛미디어(주)에 있습니다.

저작권법에 의해 보호를 받는 저작물이므로 무단 복제 및 무단 전재를 금합니다.

---

지금 하지 않으면 할 수 없는 일이 있습니다.

책으로 펴내고 싶은 아이디어나 원고를 메일([ebookwriter@hanb.co.kr](mailto:ebookwriter@hanb.co.kr))로 보내주세요.

한빛미디어(주)는 여러분의 소중한 경험과 지식을 기다리고 있습니다.

## 저자 서문

C++는 대학생 때부터 공부를 시작해서 시간 상으로는 공부한지 15년이 넘었다. 하지만 여전히 공부할 것이 있어서 난감함을 주지만 그만큼 깊게 팔 것이 많고 자유도가 높아서 내가 가장 좋아하는 프로그래밍 언어다. 프로그래밍 공부를 막 시작했을 때는 C++가 프로그래밍 언어 중 가장 인기 있는 언어였는데, 웹 시대가 시작되면서 다른 언어에 자리를 내어주고 지금은 올드한 느낌을 주는 언어가 되었다.

하지만 C++은 아직 죽은 언어가 아니다. 여전히 하드웨어 제어나 고성능 프로그램을 만들 때 주력 언어로 사용하는 등 나름 자신만의 영역을 튼튼히 가지고 있다. 특히 게임 개발에서는 C++이 많은 사랑을 받고 있다. 얼마 전까지 비디오 게임 및 온라인 게임 개발에 주로 사용되다가 요즘은 큰 인기를 얻고 있는 스마트 폰 게임 개발에서 높은 성능과 멀티 플랫폼 대응을 위해서 사용되어 오히려 이전보다 사용 영역이 더 늘어 난 것 같다.

C++이 최고 인기 프로그래밍 언어의 자리를 내주게 된 이유는 여러 가지 있지만 가장 큰 이유 중 하나가 생산성이다. 그래서 C++ 세계에서는 이 생산성이라는 부분을 어떻게 하면 향상시킬 수 있을까 고민을 하는데, 이 고민을 덜어주는 것이 바로 STL 즉 표준 템플릿 라이브러리(Standard Template Library)다.

2000년 초반만 하더라도 STL의 효용성에 대해서 찬반이 있었는데, 지금은 STL을 사용하지 않고 C++ 프로그래밍을 한다는 것은 상상하기 힘들다. 아주 특별한 경우가 아닌 이상, C++ 프로그래밍에서는 STL을 필수적으로 사용하고 있다.

STL 덕분에 C++ 프로그래밍의 난이도는 내려가고 이전에 비해서 생산성이 증가 되었다. 그래서 작년 말에 나온 C++ 새로운 표준인 C++11에서도 STL을 중요하게 여겨 다양한 기능이 추가 되었다. 앞으로 나올 새로운 표준에도 STL에 다양한 기능이 들어갈 예정이다.

글을 집필할 때, 최대한 STL의 실용적인 부분에 중점을 두었다. 기존에 나온 STL 책들은 설명이 중심이고 샘플 코드가 실제 사용하는 것과 거리감이 있어서 쉽게 와닿지 않았다. 그래서 게임 개발에서 사용했던 경험을 떠올리면서 최대한 실용적인 경우를 토대로 설명하였다. 이 책은 STL 전체를 설명하는 대신 자주 사용하고 꼭 알고 있어야 하는 것들만 추려서 설명한다. 이 책에 나온 것만 알고 있어도 STL을 사용하는 데 거의 어려움을 느끼지 않으리라 생각한다.

C++ 프로그래머라면 STL 공부는 필수입니다. 이 책을 통해서 STL을 빠르고 쉽게 공부할 수 있기 바랍니다.

끝으로 언제나 다양한 프로그래머 모임에 참여해주시는 등 나에게 큰 힘이 되어 주신 임영기님, 온라인 서버 프로그래머 커뮤니티를 만들어 주신 이지현님, 커뮤니티 운영하느라 고생 많으신 이욱진님, 게임 프로그래머가 되기 전부터 스터디를 통해서 알게 되어 지금은 같은 게임 프로그래머로서 일하면서 많은 도움을 주고 있는 조진현군에게 고마움을 전하고 싶다. 앞으로도 지금처럼 게임 업계에서 재미있는 게임을 만들기 바랍니다.

집필을 마치며,  
최홍배

## 편집자 소개

### 김제룡

훈민정음이라는 워드프로세서를 개발한 넥스소프트에서 사회생활을 시작하였다. VK MOBILE이라는 핸드폰 제작 회사에서 임베디드 시스템 개발도 해보았지만, 어려서부터 동경하던 게임개발의 꿈을 버리지 못해 게임개발자로 전향한 프로그래머다. 이스트소프트에서 카발온라인1 개발에 참여하였고, 현재는 카발온라인2 개발 및 서비스에 주력하고 있다. 외부활동보다는 조용하게 개발자의 내공을 쌓는 것과 새로운 것을 경험해보는 것을 좋아한다.

### 박민근

10년간 NC, NTL-inc, 네오위즈 게임즈 등에서 “드래곤볼 온라인”, “야구의 신” 등의 온라인 게임을 개발한 게임 프로그래머다. 현재는 새로운 모바일 회사에서 그토록 염원하던 미소녀TCG를 개발 중인 솔로 오택 프로그래머다. “게임 개발자 랩소디”라는 팟캐스트를 운영하였으며, 다양한 게임개발 관련 세미나에서 강연을 하고 있다.

### 이정재

윈도우 분야의 여러 분야를 공부하고 있는 윈도우 개발자다. 좀 더 깊고, 좀 더 많은 경험을 하지 못한 것을 아쉬워하면서 좀 더 나은 개발자가 되고 싶은 꿈을 향해 노력하고 있다. 최근에는 안드로이드 앱 개발을 하고 있다.

## 조진희

나에게 꿈을 심어준 컴퓨터와 함께 15년의 세월을 보내고 20대 마무리에 있는 연세 대학교 알고리즘연구실 대학원생이다. Positive mind로 항상 살아가기에 힘쓰며, 곧 사회에 나갈 준비를 하는 취준생이기도 하다. 30대에는 보다 다이나믹한 삶을 계획하고 있다.

## 한상곤

대학원에서 보안시스템 설계를 공부하고 있는 대학원생이다. 평소에 오픈소스 관련 프로젝트를 친구들과 함께 신나게 진행하고 있으며 현재는 파이썬에 관심을 가지고 공부하고 있다. 오픈소스도 관심이 많아서 우분투 관련 모임에 꾸준히 참여하고 있다.

## 편집을 마치며

요즘 전 세계에는 스마트폰의 광풍이 불고 있습니다. 안드로이드와 아이폰의 전쟁이라고 할 수 있는 흐름이 있어서, 안드로이드를 개발하기 위한 자바와 아이폰을 개발하기 위한 오브젝티브 C 언어는 마치 IT개발의 세계를 양분하는 프로그래밍 언어의 전부인듯한 느낌을 받기도 합니다.

STL은 C++을 사용하는 많은 분들이 사용하는 라이브러리입니다. STL 도서의 필요성을 의심하지는 않지만 스마트폰이 유행하는 이 시점에서 STL에 대한 책을 낸다고 했을 때, 시기가 맞지 않는 것은 아닐까라는 생각이 들기도 했습니다.

그러나 예전에 이 책의 근간이 된 인터넷 문서를 통해 STL에 대한 이해를 넓히고, 좀 더 친근하게 다가갈 수 있었던 경험이 생각났고, 제가 받았던 도움과 경험을 다른 사람들에게 나누어 줄 수 있는 기회라고 생각되어 편집에 참여하게 되었습니다. 편집을 진행하는 중에는 저자의 의도와 역량을 제대로 전달하기 위해 노력을 했지만, 의외로 고려할 많은 것이 있다는 것을 알 수 있었습니다. 우리가 읽는 한 권의 책이 나오기 위해서는 저자의 역할과 노력이 많다는 것은 알고 있었지만, 그것 말고도 편집의 어려움과 많은 작업을 직접 경험해 볼 수 있었습니다.

앞으로 책을 읽을 때, 책이 전하는 내용뿐만 아니라 한 권의 책이 나오는데 노력했던 편집자를 포함한 많은 분들의 노고를 한 번쯤 되새기면서 책을 읽게 되지 않을까 생각합니다. 책이 만들어지는 과정을 통해 책의 소중함을 다시 한번 생각할 수 있었던 계기가 되었습니다.

함께 작업했었던 여러 분들(김제룡 님, 박민근 님, 조진희 님, 한상곤 님)과 좋은 기회를 제공해 주신 김병희 대리님과 김창수 팀장님께 감사 드립니다.

편집자 대표

이정재



# 대상 독자

초급

초중급

중급

중고급

고급

이 책은 STL의 기본적인 개념을 이해하고 사용법을 알고 싶은 개발자를 대상으로 한다. 때문에 독자가 C++의 기본적인 문법과 컴파일러를 사용하는 방법에 익숙하다고 가정한다. 즉, C++의 문법과 컴파일러를 통해 소스코드를 빌드하여 결과를 보는 방법은 논하지 않고 오직 STL의 기본적인 개념을 이해시키고, STL의 사용법을 보여주는 데만 중점을 둘 것이다.

# 한빛전자책 알림

한빛 eBook 리얼타임은 IT 개발자를 위한 eBook 입니다. 요즘 IT 업계에는 하루가 멀다 하고 수많은 기술이 나타나고 사라져 갑니다. 인터넷을 아무리 뒤져도 조금이나마 정리된 정보를 찾는 것도 쉽지 않습니다. 또한 잘 정리되어 책으로 나오기까지는 오랜 시간이 걸립니다. 어떻게 하면 조금이라도 더 유용한 정보를 빠르게 얻을 수 있을까요? 어떻게 하면 남보다 조금 더 빨리 경험하고 습득한 지식을 공유하고 발전시켜 나갈 수 있을까요? 세상에는 수많은 종이책이 있습니다. 그리고 그 종이책을 그대로 옮긴 전자책도 많습니다. 전자책에는 전자책에 적합한 콘텐츠와 전자책의 특성을 살린 형식이 있다고 생각합니다.

한빛이 지금 생각하고 추구하는, 개발자를 위한 리얼타임 전자책은 이렇습니다.

## 1. eBook Only - 빠르게 변화하는 IT 기술에 대해 핵심적인 정보를 신속하게 제공합니다.

500페이지 가까운 분량의 잘 정리된 도서(종이책)가 아니라, 핵심적인 내용을 빠르게 전달하기 위해 조금은 거칠지만 100페이지 내외의 전자책 전용으로 개발한 서비스입니다. 독자에게는 새로운 정보를 빨리 얻을 수 있는 기회가 되고, 자신이 먼저 경험한 지식과 정보를 책으로 펴내고 싶지만 너무 바빠서 엄두를 못 내시는 선배, 전문가, 고수분에게는 보다 쉽게 집필하실 기회가 되리라 생각합니다. 또한 새로운 정보와 지식을 빠르게 전달하기 위해 O'Reilly의 전자책 번역 서비스도 준비 중이며, 조만간 선보일 예정입니다.

## 2. 무료로 업데이트되는, 전자책 전용 서비스입니다.

종이책으로는 기술의 변화 속도를 따라잡기가 쉽지 않습니다. 책이 일정한 분량 이상으로 집필되고 정리되어 나오는 동안 기술은 이미 변해 있습니다. 전자책으로 출간된 이후에도 버전 업을 통해 중요한 기술적 변화가 있거나, 저자(역자)와 독자가 소통하면서 보완되고 발전된 노하우가 정리되면 구매하신 분께 무료로 업데이트해 드립니다.

### 3. 독자의 편의를 위하여, DRM-Free로 제공합니다.

구매한 전자책을 다양한 IT기기에서 자유롭게 이용하실 수 있도록 DRM-Free PDF 포맷으로 제공합니다. 이는 독자 여러분과 한빛이 생각하고 추구하는 전자책을 만들어 나가기 위해, 독자 여러분이 언제 어디서 어떤 기기를 사용하시더라도 편리하게 전자책을 보실 수 있도록 하기 위함입니다.

### 4. 전자책 환경을 고려한 최적의 형태와 디자인에 담고자 노력했습니다.

종이책을 그대로 옮겨 놓아 가독성이 떨어지고 읽기 힘든 전자책이 아니라, 전자책의 환경에 가능한 최적화하여 쾌적한 경험을 드리고자 합니다. 링크 등의 기능을 적극적으로 이용할 수 있음은 물론이고 글자 크기나 행간, 여백 등을 전자책에 가장 최적화된 형태로 새롭게 디자인하였습니다.

앞으로도 독자 여러분의 충고에 귀 기울이며 지속해서 발전시켜 나가도록 하겠습니다.

지금 보시는 전자책에 소유권한을 표시한 문구가 없거나 타인의 소유권한을 표시한 문구가 있다면 위법하게 사용하고 계실 가능성이 높습니다. 이 경우 저작권법에 의해 불이익을 받으실 수 있습니다.

구입하신 도서는 사본을 보관하지 않는다는 조건으로 다양한 기기에 사용할 수 있습니다. 또한 한빛미디어 사이트에서 구입하신 후에는 횡수에 관계없이 다운받으실 수 있습니다.

한빛미디어 전자책은 인쇄, 검색, 복사하여 붙이기가 가능합니다.

전자책은 오탈자 교정이나 내용의 수정보완이 이뤄지면 업데이트 관련 공지를 이메일로 알려드리며, 구매하신 전자책의 수정본은 무료로 내려받으실 수 있습니다.

이런 특별한 권한은 한빛미디어 사이트에서 구입하신 독자에게만 제공되며, 다른 사람에게 양도나 이전되지 않습니다.

# 차례

01	<b>C++ STL 소개</b>	1
----	-------------------	---

---

1.1	STL이 무엇인지 알고 있는가? .....	1
1.2	STL은 어떻게 만들었을까? .....	1
1.3	언어를 공부한 사람은 템플릿에 대해 잘 알고 있을까? .....	1
1.4	객체 지향 프로그래밍(OOP) C++ .....	2
1.5	Generic Programming이라는 것을 들어 보았는가? .....	3
1.6	대체 C++언어에서 무엇을 '총칭'화 할까? .....	4

02	<b>함수 템플릿</b>	5
----	---------------	---

---

2.1	두 값을 비교하는 함수를 만들어야 한다. ....	5
2.2	Max 함수를 하나로 만들고 싶다. 어떻게 해야 할까? .....	7
2.3	함수 템플릿과 컴파일.....	9
2.4	Max 함수 템플릿에 개선점이 있을까? .....	11
2.5	class T 라는 것을 본적이 있나요? .....	11
2.6	이제 Max 함수 템플릿에는 문제가 없을까? .....	12
2.7	typename을 하나가 아닌 복수 개 사용하면 된다. ....	13
2.8	함수 템플릿의 전문화 라는 것이 있다. ....	14
2.9	난-타입(non-type) 함수 템플릿 .....	16

03	<b>클래스 템플릿</b>	19
----	----------------	----

---

3.1	경험치 변경 이력 저장 .....	19
3.2	게임 돈 변경 이력도 저장해 주세요.....	22

3.3	클래스 템플릿을 사용하는 방법 .....	25
3.4	Stack 템플릿 클래스 .....	25
3.5	클래스 템플릿에서 non-type 파라미터 사용 .....	29
3.6	템플릿 파라미터 디폴트 값 사용 .....	32
3.7	스택 클래스의 크기를 클래스 생성자에서 지정 .....	33
3.8	클래스 템플릿 전문화 .....	36
3.9	클래스 템플릿 부분 전문화 .....	42
3.10	싱글톤 템플릿 클래스 .....	45
3.11	클래스 템플릿 코딩 스타일 개선 .....	47
3.12	클래스 선언과 정의를 각각 다른 파일에 하려면 .....	50

04

## 연결 리스트

53

4.1	list의 자료구조 .....	53
4.2	연결 리스트의 특징 .....	53
4.3	STL list를 사용하면 좋은 점 .....	55
4.4	list 사용방법 .....	57
4.5	list를 사용한 스택 .....	81
4.6	과제 .....	86

05

## 벡터(vector)

88

5.1	vector의 자료구조 .....	88
5.2	배열의 특징 .....	88
5.3	vector를 사용해야 하는 경우 .....	90
5.4	vector vs. list .....	90
5.5	vector 사용방법 .....	92
5.6	vector의 주요 멤버들 .....	92
5.7	과제 .....	113

06	<b>덱(deque)</b>	<b>115</b>
6.1	deque의 자료구조 .....	115
6.2	Deque의 특징.....	116
6.3	deque을 사용하는 경우 .....	117
6.4	deque vs. vector.....	118
6.5	deque 사용방법 .....	119
6.6	과제 .....	138
07	<b>해시 맵(hash_map)</b>	<b>139</b>
7.1	시퀀스 컨테이너와 연관 컨테이너.....	139
7.2	연관 컨테이너로는 무엇이 있을까? .....	140
7.3	hash_map의 자료구조 .....	141
7.4	hash_map을 사용할 때와 사용하지 않을 때 .....	142
7.5	hash_map 사용방법 .....	143
08	<b>맵(map)</b>	<b>155</b>
8.1	map의 자료구조 .....	155
8.2	트리 자료구조의 특징.....	156
8.3	map을 언제 사용해야 될까? .....	156
8.4	map 사용방법 .....	156
8.5	과제 .....	165
09	<b>셋(set)</b>	<b>166</b>
9.1	set 이란.....	166
9.2	set을 사용할 때 .....	166
9.3	set 사용방법 .....	167

9.4	과제 .....	178
-----	----------	-----

---

10.1	STL 알고리즘 분류.....	179
10.2	조건자.....	180
10.3	변경 불가 시퀀스 알고리즘 .....	180
10.4	변경 가능 시퀀스 알고리즘 .....	188
10.5	정렬 관련 알고리즘 .....	199
10.6	범용 수치 알고리즘 .....	210

# 1 C++ STL 소개

## 1.1 STL이 무엇인지 알고 있는가?

C++를 주 프로그래밍 언어로 사용하고 있다면 알고 있으리라 생각한다. STL은 C++ 언어의 '표준 템플릿 라이브러리' [Standard Template Library](#)의 약자다. STL을 간단하게 말하자면 일반적으로 많이 사용되는 자료구조와 알고리즘을 모은 라이브러리다.

STL은 C++ 언어가 처음 만들어질 때부터 있었던 것이 아니라 C++ 표준이 정해지기 전인 1993년 말 무렵에 알렉스 스테파노브 [Alex Stepanov](#)가 C++ 언어의 창시자인 비얀 스트로스트룹 [Bjarne Stroustrup](#)에게 제안 후 준비 기간을 걸쳐서 1994년에 표준 위원회에서 초안이 통과되었다(참고로 C++ 표준(C++98)은 1989년에 시작되어 1998년 9월에 마무리되었다).

## 1.2 STL은 어떻게 만들었을까?

물음에 대한 답은 STL의 실제 이름에 포함되어 있다. 좀 더 힌트를 준다면, STL을 구성하는 세 개의 단어 중에서 중간에 있는 단어를 잘 살펴보면 된다. 짐작이 가는가?

정답은 중간에 있는 템플릿 [Template](#)이다. STL을 이해하려면, STL을 구성하고 있는 C++의 템플릿을 반드시 이해해야 한다. 템플릿은 C++를 더욱 강력하게 사용하는 데 꼭 필요하다.

## 1.3 언어를 공부한 사람은 템플릿에 대해 잘 알고 있을까?

예전에 C++을 잠시 공부했거나 지금 C++을 공부하고 있더라도 C++ 관련 서적을 한 권 혹은 두 권 정도 읽어본 경우라면 템플릿이라는 단어가 생소할 수 있다.

위에서 언급했듯이 템플릿은 C++이 세상에 나오면서 같이 나온 것이 아니라 1994년 무렵에야 세상에 조금씩 소개되다가 1998년에 C++ 표준이 정립된 이후 C++ 언어의 한 부분으로서 인정되었다.



1994년까지는 템플릿을 지원하는 C++ 컴파일러가 없었고, Microsoft의 C++ 톨로 유명한 Visual C++도 버전 6에서도 템플릿을 완벽하게 지원하지 못했다. 템플릿은 Visual Studio .NET 2003에서부터 제대로 지원되었다(아직도 템플릿 기능을 100% 완벽하게 지원하지는 못한다).

2000년 이전에 나온 C++ 입문서를 보면 템플릿에 대하여 빠뜨린 것이 꽤 많다. 요즘 나오는 입문서도 템플릿 부분이 빠져 있기도 하다. 템플릿은 일반 C++ 입문서에서는 가장 뒷부분에 나오다 보니 공부를 하다가 중간에 포기하게 되면 클래스라는 것은 알아도 템플릿은 잘 모를 수 있다.

개인적으로 C 언어를 생각하면 포인터가 떠오르고, C++ 언어를 생각하면 클래스와 템플릿이 떠오른다. 이유는 C 언어나 C++ 언어를 배울 때 정확하게 이해하기 가장 어려웠던 것이고 필자가 배웠던 다른 언어들에 비해 크게 달랐던 것이기 때문이다.

특히 C언어의 포인터는 처음 배울 때 문법적인 사용방법이 잘 이해가 안 돼서 어려웠지만, C++의 클래스나 템플릿은 문법적인 사용방법이 어려운 것이 아니고 프로그램 설계와 관련해서 클래스와 템플릿을 어떻게 활용해야 하는지 이해하기 어려웠다.

## 1.4 객체 지향 프로그래밍(OOP) C++

C++ 언어를 소개할 때 가장 먼저 이야기하는 것이 객체지향이라는 것이다. 현대의 많은 프로그래밍 언어들은 객체 지향 프로그래밍(OOP(Object-Oriented Programming))을 지원합니다.

C 언어와 C++ 언어는 이름이 비슷하듯이 유사한 부분도 많다. C 언어로 프로그래밍 할 때는 절차 지향 프로그래밍을 하게 된다. C++도 절차 지향 프로그래밍을 할 수 있다. 그러나 제대로 된 C++ 프로그래밍을 하려면 객체 지향 프로그래밍을 해야 한다. 보통 C 언어를 배운 후 바로 이어서 C++를 배울 때는 객체 지향 프로그래밍에 대한 이해가 부족하다. 그래서 C 언어로 프로그래밍 할 때와 같은 절차 지향 프로그래밍을 하여 이른바 'C++를 가장한 C 프로그래밍'을 한다는 소리를 듣기도 한다. C++ 언어로 객체 지향 프로그래밍을 할 수 있는 것은 C 언어에는 없는 클래스가 있기 때문이다.

**질문:** C++로 할 수 있는 프로그래밍 스타일은 절차적 프로그래밍, 객체 지향 프로그래밍만 있을까?

**답변:** 아니다. Generic Programming 도 가능하다.

## 1.5 Generic Programming이라는 것을 들어 보았는가?

필자가 프로그래밍을 배울 때는 일반적으로 C++ 언어를 배우기 전에 C 언어를 공부했다. C 언어를 처음 공부했던 시기가 대략 1994년 즈음 이다. 그 당시의 다른 초보 프로그래머들처럼 포인터의 벽에 부딪혀 좌절하고, 도망(?)가서 3D Studio라는 그래픽 툴을 공부하다가 내가 할 것이 아니라는 생각에 포기하고, 1995년에 다시 C 언어를 공부하였고 이후 바로 C++ 언어를 공부했다.

이때도 OOP라는 단어는 무척 자주 들었고 C++로 프로그래밍을 잘한다는 것은 OOP를 잘한다는 것과 같은 뜻이었다.

대학을 다닐 때부터 내 용돈의 많은 부분은 프로그래밍 책을 사는 데 사용되었다. 그중에서 C++ 언어 책을 꽤 많이 구매하여 보았다(다만, 제대로 이해한 책은 별로 없었다).

책에서는 언제나 OOP라는 단어는 무수히 많이 보았지만, Generic Programming이라는 단어를 그 당시에 본 기억이 없다. 필자가 Generic Programming이라는 단어를 알게 된 것은 2001년 무렵이다. C++ 언어를 공부한지 거의 6년이 될 무렵에 알게 되었다.

아마 지금 공부하는 분들도 Generic Programming이라는 단어는 좀 생소할 것이다. Generic Programming은 한국에서는 보통 '일반적 프로그래밍'이라고 이야기 한다. 필자도 처음에는 그렇게 들었다.

그러나 이것은 잘못된 표현이라 생각한다. 영어 사전을 보면 Generic 이라는 것은 '총칭(總稱)적인' 이라는 뜻도 있는데 이것이 '일반적'이라는 단어보다 더 확실하며 필자가 2004년에 일본에서 구입한 『C++ 설계와 진화』(Bjarne Stroustrup 저)라는 일본 번역서에도 Generic은 총칭으로 표기하고 있다.

그럼 Generic Programming은 무엇일까? 네이버 사전에서 Generic이라는 단어를 검색하면, 다음과 같은 부분을 볼 수 있다.

【문법】 총칭적인

the generic singular 총칭 단수 《보기:The cow is an animal.》

보기의 영문을 필자의 짧은 영어 실력으로 번역을 하면 '암소는 동물이다'이다. 소는 분명히 고양이나 개와는 다르지만 '동물'이라는 것으로 „총칭“할 수 있다.

## 1.6 대체 C++언어에서 무엇을 '총칭'화할까?

필자가 만드는 프로그램은 Windows 플랫폼에서 실행되는 '온라인 게임 서버' 프로그램이다. 온라인 게임 서버를 만들 때는 „어떤 기능이 있어야 되는가?“를 결정한 후 클래스를 만든다. 클래스는 아는 바와 같이 멤버 변수와 멤버 함수로 이루어져 있다. 그리고 멤버 함수도 그 내용은 필자의 생각에 의해 변수들을 조작하는 내용으로 되어 있다.

'암소는 동물이다'라는 식으로 C++ 언어에서 총칭을 하는 것은 변수의 타입<sup>type</sup>을 총칭화 하는 것이다. 변수의 타입을 총칭화하면 다음과 같은 장점이 있다.

- 템플릿을 이용하면 총칭화된 타입을 사용하는 클래스와 함수를 만들 수 있다.
- 템플릿을 사용하면 타입에 제약을 받지 않는 로직을 기술할 수 있다.

그리고 Generic Programming을 하기 위해서는 템플릿이 꼭 필요하다. 그런데 STL이 무엇으로 만들어졌는가? 바로, 템플릿으로 만들어졌다. STL은 Generic Programming으로 만들어진 가장 대표적인 예다.

필자 나름대로 템플릿을 이해하는 데 도움이 되었으면 해서 이런저런 이야기를 했는데 과연 도움이 되었는지 모르겠다. 아마 설명만 듣고서는 템플릿에 대해 명확하게 이해를 하지 못하리라 생각한다. 우리 프로그래머들은 정확하게 이해하려면 코드를 봐야겠지요? 템플릿은 크게 함수 템플릿과 클래스 템플릿으로 나눌 수 있다.

## 2 함수 템플릿

### 2.1 두 값을 비교하는 함수를 만들어야 한다

앞서 필자가 하는 일을 이야기했다. 온라인 게임을 만들고 있다. 게임에서 구현해야 되는 것에는 캐릭터 간에 체력(HP)을 비교하는 것이 필요하다. 그래서 두 개의 int 타입을 비교하는 Max라는 이름의 함수를 하나 만들었다.

```
int Max( int a, int b );
```

일을 다 끝낸 후 다음 기획서를 보니 캐릭터와 NPC(Non Player Character)가 전투를 하는 것을 구현해야 하는데 여기에는 경험치를 비교하는 기능이 필요하다. 구현해야 하는 것은 위에서 만든 Max 함수와 같다. 그래서 그것을 사용했다.

#### [리스트 2-1]

---

```
#include <iostream>
using namespace std;

int Max( int a, int b )
{
    return a > b ? a : b;
}

void main()
{
    int Char1_HP = 300;
    int Char2_HP = 400;
    int MaxCharHP = Max( Char1_HP, Char2_HP );
    cout << "HP 중 가장 큰 값은 " << MaxCharHP << "입니다." << endl << endl;

    float Char1_Exp = 250.0f;
    float Char2_Exp = 250.57f;
    float MaxCharExp = Max( Char1_Exp, Char2_Exp );
    cout << "경험치 중 가장 큰 값은 " << MaxCharExp << "입니다." << endl << endl;
}
```

---

앗, 체력(HP)을 저장하는 변수의 타입은 int인데, 경험치를 저장하는 변수의 타입은 int가 아닌 float 타입이다.

HP 중 가장 큰 값은 400입니다.

경험치 중 가장 큰 값은 250입니다.

당연하게 경험치를 비교하는 부분은 버그가 있다. 앞에 만들었던 Max와는 다르게 비교하는 변수의 타입이 float인 것이 필요하여 새로 만들었다.

## [리스트 2-2]

---

```
float Max( float a, float b )
{
    return a > b ? a : b;
}
```

---

함수 오버로딩에 의해 경험치를 비교할 때는 int 타입의 Max가 아닌 float 타입을 비교하는 Max가 호출되어 버그가 사라지게 되었다.

이제 경험치 비교는 끝나서 다음 기획서에 있는 것을 구현해야 한다. 이번에는 돈을 비교하는 것이 있다. 그런데 돈을 저장하는 변수의 타입은 \_\_int64입니다. \_\_int64는 비주얼 C++에서만 사용할 수 있는 64비트 정수 타입이다. \_\_int64 타입을 비교하는 것은 앞에서 만든 int 타입의 Max나 float 타입의 Max로 할 수 없다. 함수에서 사용하는 변수의 타입만 다를 뿐 똑같은 것을 또 만들어야 한다.

---

```
__int64 Max(__int64 a, __int64 b )
{
    return a > b ? a : b;
}
```

---

현재까지만 하더라도 이미 똑같은 로직으로 구현된 함수를 3개나 만들었는데, 게임에서 사용하는 캐릭터의 정보는 HP, 경험치, 돈 이외에도 더 많다. 저는 앞으로 Max 함수를 몇 개 더 만들어야 할지 모른다. Max 함수의 구현을 고쳐야 한다면 모든 Max 함수를 찾아야 한다. 함수 오버로딩은 문제를 해결하지만, 코

드가 커지고 유지보수는 어렵게 만든다.

프로그래밍에서 유지보수는 아주 중요하다. 왜냐하면, 프로그래밍은 언제나 변경이 가해지기 때문이다. 유지보수를 편하게 하는 가장 간단한 방법은 유지보수할 것을 줄이는 것이다.

## 2.2 Max 함수를 하나로 만들고 싶다. 어떻게 해야 할까?

앗, 혹시 모른다고요? 필자가 이 앞에 템플릿에 대해 설명할 때 이런 말을 하지 않았나요?

'템플릿을 사용하면 타입에 제약을 받지 않는 로직을 기술할 수 있다'

즉, 템플릿을 사용하면 된다.

### 2.2.1 함수 템플릿 Max를 만들자

다음 코드는 템플릿을 사용하여 Max 함수를 구현한 것이다.

[리스트 2-3]

---

```
#include <iostream>
using namespace std;

// 템플릿으로 만든 값을 비교하는 Max 함수
template <typename T> T Max(T a, T b )
{
    return a > b ? a : b;
}

void main()
{
    int Char1_HP = 300;
    int Char2_HP = 400;
    int MaxCharHP = Max( Char1_HP, Char2_HP );
    cout << "HP 중 가장 큰 값은" << MaxCharHP << "입니다." << endl << endl;
```

---

```

float Char1_Exp = 250.0f;
float Char2_Exp = 250.57f;
float MaxCharExp = Max( Char1_Exp, Char2_Exp );
cout << "경험치 중 가장 큰 값은" << MaxCharExp << "입니다." << endl << endl;
}

```

[리스트 2-3]의 실행 결과는 다음과 같다.

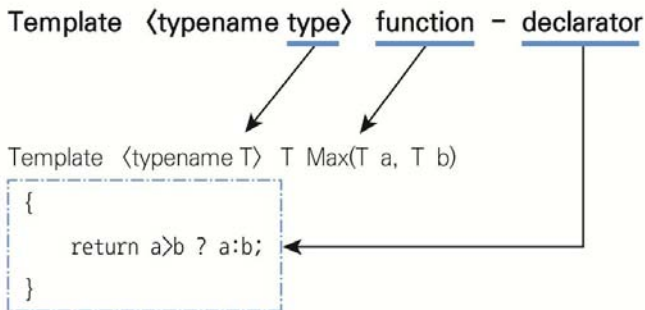
```

HP 중 가장 큰 값은 400입니다.
경험치 중 가장 큰 값은 250입니다.

```

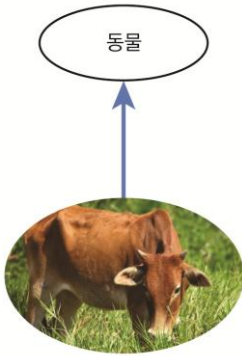
이번에는 경험치 비교가 정확하게 이루어졌다. 템플릿을 사용하게 되어 이제는 불필요한 Max 함수를 만들지 않아도 된다. [리스트 2-3] 코드에서 template으로 만든 함수를 '함수 템플릿'이라고 한다. 함수 템플릿을 정의하는 방법과 같다.

그림 2-1 함수 템플릿 정의



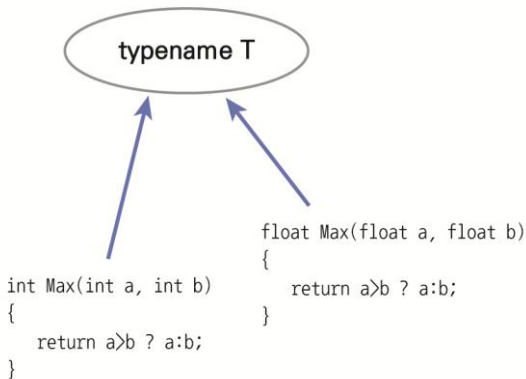
‘템플릿을 사용하면 Generic Programming을 할 수 있다’ 라고 앞서 이야기 했는데 위의 Max 함수 템플릿을 보고 좀 이해를 할 수 있었는가? 혹시나 해서 그림 2-2로 조금만 더 설명한다.

그림 2-2 Max 함수 템플릿



암소를 충청<sup>Generic</sup>화하면 동물이라고 할 수 있다. Max 함수 템플릿에서는 함수의 반환 값과 함수 인자인 a 와 b의 타입인 int나 float를 T로 Generic화했다.

그림 2-3 그림 제목



## 2.3 함수 템플릿과 컴파일

하나의 Max 함수 템플릿을 만들었는데 어떻게 int 타입의 Max와 float 타입의 Max를 사용할 수 있을까? 비밀은 컴파일하는 과정에 있다. 컴파일할 때 템플릿



으로 만든 것은 템플릿으로 만든 함수를 호출하는 부분에서 평가한다. 가상 함수처럼 실행시간에 평가하는 것이 아니다.

컴파일을 할 때, 함수 템플릿을 평가하므로 프로그램의 성능을 떨어뜨리지 않는다. 컴파일할 때 평가를 하면서 문법적으로 에러가 없는지 검사한다. 만약 에러가 있다면 컴파일 에러를 출력한다. 에러가 없다면 관련 코드를 내부적으로 생성한다.

[리스트 2-3]을 예로 들면, void main()의 다음 부분을 컴파일하면 Max를 호출할 때 사용한 인자의 변수의 타입이 Max에서 정의 한 문법에 틀리지 않는지 체크한 후 int 타입을 사용하는 Max 함수의 코드를 만든다.

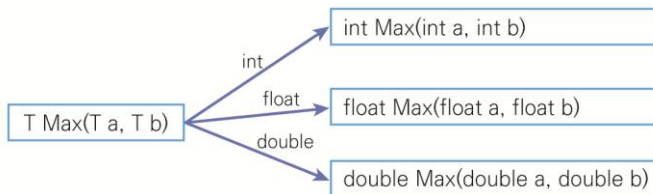
```
int MaxCharHP = Max( Char1_HP, Char2_HP );
```

이후 다음 부분에서 Max를 만나면 이번에도 위의 int 때와 같이 문법 체크를 한 후 에러가 없다면 float를 사용하는 Max 함수 코드를 만든다.

```
float MaxCharExp = Max( Char1_Exp, Char2_Exp );
```

Max가 만들어지는 과정을 나타내면 아래와 같다. 모든 타입에 대해 Max 함수를 만드는 것은 아니다. 코드에서 사용한 타입에 대해서만 Max 함수가 만들어진다.

그림 2-4 그림 제목



참고로 이렇게 만들어지는 코드는 소스 코드에 만들어지는 것이 아니고 프로그램의 코드 영역에 만들어진다. 컴파일 타임에 함수 템플릿을 평가하고 관련 코드를 만들기 때문에 템플릿을 많이 사용하면 컴파일 시간이 길어질 수 있으며, 각 타입에 맞는 코드를 만들어내므로 실행 파일의 크기도 커질 수 있다.

## 2.4 Max 함수 템플릿에 개선점이 있을까?

힌트를 준다면 Max의 두 인자 값은 함수 내부에서 변경되지 않는다. 그리고 인자의 타입은 C++의 기본형뿐만이 아닌 크기가 큰 타입을 사용할 수도 있다.

생각이 났는가? C++ 기초 공부를 차근차근 쌓아 올렸다면 알아차렸으리라 생각한다.

정답은 Max 함수 템플릿을 만들 때 템플릿의 인자에 const와 참조를 사용하는 것이다. Max 함수는 함수의 내부에서 함수의 인자를 변경하지 않는다. 그러니 함수에 const를 사용하여 내부에서 변경하는 것을 명시적으로 막고 Max 함수를 사용하는 사람에게 알리는 역할을 한다.

C++에서 함수 인자의 전달을 빠르게 하는 방법은 참조로 전달하는 것이다. 위의 Max 함수는 int나 float 같은 크기가 작은 타입을 사용하였기 때문에 참조로 전달하는 것이 큰 의미는 없지만, 만약 구조체나 클래스로 만들어진 크기가 큰 변수를 사용할 때는 참조로 전달하는 것이 훨씬 빠르다. 앞에 만든 Max 함수 템플릿을 const와 참조를 사용하는 것으로 바꾸어 보았다.

### [리스트 2-4]

---

```
template <typename T> const T& Max(const T& a, const T& b)
{
    return a > b ? a : b;
}
```

---

## 2.5 class T 라는 것을 본적이 있나요?

함수 템플릿을 만들 때 'typename'을 사용했다. 그러나 좀 오래된 C++ 책에서 템플릿에 대한 글을 본 적이 있다면 'class'를 사용한 것도 본 적이 있을 것이다.

### [리스트 2-5]

---

```
template <class T> const T& Max(const T& a, const T& b)
{
    return a > b ? a : b;
}
```

---

typename과 class는 기능적으로 다른 것이 아니다. 템플릿이 표준이 되기 전에는 'class'를 사용했다. 그래서 표준화 이전이나 조금 지난 뒤에 나온 책에서는 'class'로 표기했다. 그리고 예전에 만들어진 C++ 컴파일러도 템플릿 인자 선언으로 'class'만 지원했다. 만약, C++ 표준화 전후에 만들어진 컴파일러에서는 'class'를 사용해야 한다.

현재의 컴파일러에서는 'class', 'typename' 둘 다 지원한다. 하지만, 'class'보다 프로그래머에게 '타입'을 추상화한 것이라는 의미 전달을 명확하게 하는 typename을 사용한다. class만 지원하는 오래된 C++ 컴파일러에서 컴파일 해야 하는 것이 아니면 꼭 'typename'을 사용하도록 한다.

## 2.6 이제 Max 함수 템플릿에는 문제가 없을까?

위에서 Max 함수 템플릿에 대해서 const와 참조로 개선을 했는데 이제 문제가 없을까? 그럼 다음 코드는 문제가 없이 컴파일이 잘 될까?

### [리스트 2-6]

---

```
// [리스트 2-3]의 Max 함수 템플릿을 사용한다.
void main()
{
    int Char1_MP = 300;
    double Char1_SP = 400.25;
    double MaxValue1 = Max( Char1_MP, Char1_SP );
    cout << "MP와 SP 중 가장 큰값은" << MaxValue1 << "입니다." << endl << endl;

    double MaxValue2 = Max( Char1_SP, Char1_MP );
    cout << "MP와 SP 중 가장 큰값은" << MaxValue2 << "입니다." << endl << endl;
}
```

---

[리스트 2-6]을 컴파일 하면 다음과 같은 에러가 출력된다.

---

```
max.cpp
max.cpp(16) : error C2782: 'const T &Max(const T &,const T &)'
           : 템플릿 매개 변수 'T'이(가) 모호합니다.
max.cpp(6) : 'Max' 선언을 참조하십시오.
'double'일 수 있습니다.
```

---

---

```
또는      'int'
max.cpp(19) : error C2782: 'const T &Max(const T &,const T &)'
           : 템플릿 매개 변수 'T'이(가) 모호합니다.
max.cpp(6) : 'Max' 선언을 참조하십시오.
'int'일 수 있습니다.
또는      'double'
```

---

이유는 컴파일러는 사람이 아니어서 서로 다른 타입의 인자가 들어오면 템플릿의 파라미터 T를 사용한 함수의 인자 a와 b의 타입을 int로 해야 할지, double로 해야 할지 판단할 수가 없기 때문이다. 이 문제는 어떻게 해결 해야 할까?

## 2.7 typename을 하나가 아닌 복수 개 사용하면 된다

위의 문제는 Max 함수를 정의할 때 typename을 하나만 사용해서 타입을 하나만 선언했다. 이제 typename을 여러 개 사용하면 위의 문제를 풀 수 있다.

### [리스트 2-7]

---

```
template <typename T1, typename T2> const T1& Max(const T1& a, const T2& b )
{
    return a > b ? a : b;
}
```

---

[리스트 2-7]의 함수 템플릿을 사용하면 Max 함수의 인자 타입을 int와 double 혹은 double과 int 타입을 사용해도 컴파일 이 잘된다. 그럼 제대로 실행되는지 실행해보자.

```
MP와 SP 중 가장 큰값은400입니다.
MP와 SP 중 가장 큰값은400.25입니다.
```

앗, 실행 결과에 오류가 있다.

---

```
int Char1_MP = 300;
double Char1_SP = 400.25;
double MaxValue1 = Max( Char1_MP, Char1_SP );
```

---

이 코드는 300과 400.25를 비교한다. 결과는 400.25가 나와야 하는데 400이 나와버렸다.

이유는 [리스트 2-7]의 함수 템플릿의 반환 값으로 T1을 선언했기 때문에 int 타입과 double 타입을 순서대로 함수 인자에 사용하면 반환 값의 타입이 int형으로 되어 버리기 때문이다. 이렇게 서로 다른 타입을 사용하는 경우에는 반환 값을 아주 조심해야 한다. 그리고 위의 예에서는 함수 템플릿의 파라미터로 typename을 2개 사용했지만 그 이상도 사용할 수 있다.

위의 Max 함수 템플릿 만족스러운가? 필자는 웬지 아직도 좀 불 만족스럽다.

```
Max(int, double);
```

여기서는 실수를 하면 찾기 힘든 버그가 발생할 확률이 높다. 이것을 어떻게 풀어야 될까?

## 2.8 함수 템플릿의 전문화 라는 것이 있다.

Max(int, double)을 사용하면 Max 함수 템플릿이 아닌 이것에 맞는, 특별하게 만든 함수를 사용하도록 한다. 함수 템플릿의 전문화 [Specialization](#)라는 특별한 상황에 맞는 함수를 만들면 함수 오버로드와 같이 컴파일러가 상황에 맞는 함수를 선택하도록 한다.

### [리스트 2-8]

---

```
#include <iostream>
using namespace std;

// 템플릿으로 만든값을비교하는Max 함수
template <typename T1, typename T2> const T1& Max(const T1& a, const T2& b )
{
    cout << "Max(const T& a, const T& b) 템플릿 버전 사용" << endl;
    return a > b ? a : b;
}

// 전문화시킨Max 함수
template <> const double& Max(const double& a, const double& b)
```

---

---

```

{
    cout << "Max(const double& a, const double& b) 전문화 버전 사용" << endl;
    return a > b ? a : b;
}

void main()
{
    double Char1_MP = 300;
    double Char1_SP = 400.25;
    double MaxValue1 = Max( Char1_MP, Char1_SP );
    cout << "MP와 SP 중 가장 큰 값은" << MaxValue1 << "입니다." << endl << endl;

    int Char2_MP = 300;
    double Char2_SP = 400.25;
    double MaxValue2 = Max( Char2_MP, Char2_SP );
    cout << "MP와 SP 중 가장 큰 값은" << MaxValue2 << "입니다." << endl << endl;
}

```

---

[리스트 2-8]의 코드를 실행한 결과는 다음과 같다.

```

Max(const double& a, const double& b) 전문화 버전 사용
MP와 SP 중 가장 큰 값은400.25입니다.

Max(const T& a, const T& b) 템플릿 버전 사용
MP와 SP 중 가장 큰 값은400입니다.

```

컴파일러는 프로그래머의 생각을 완전히 이해하지는 않는다. 그래서 컴파일러가 어떠한 것을 선택할지 이해하고 있어야 된다. [리스트 2-8]은 double에 전문화된 Max 함수를 만든 예다.

**질문** Max(10.1, 20.4)를 호출한다면 Max(T, T)가 호출 될까? 아님 Max(double, double)가 호출 될까?

답을 빨리 알고 싶을 테니 뜬 들이지 않고 결과를 바로 보자.

`Max(const double& a, const double& b)` 전문화 버전 사용

전문화 버전이 호출 되었다. 이유는 호출 순서에 규칙이 있기 때문이다(최선에서 최악으로). 호출 순서는 다음과 같다.

1. 전문화된 함수와 맞는지 검사한다.
2. 템플릿 함수와 맞는지 검사한다.
3. 일반 함수와 맞는지 검사한다.

위의 순서를 잘 기억하고 전문화 함수를 만들어야 한다. 잘못하면 찾기 힘든 버그를 만들 수가 있다. 이제 함수 템플릿에 대한 이야기는 거의 다 끝난 것 같다.

아... 하나 더 있다. 지금까지 살펴본 것은 타입만을 템플릿 파라미터로 사용했는데 꼭 타입만 함수 템플릿에 사용할 수 있는 것은 아니다.

## 2.9 난-타입(non-type) 함수 템플릿

온라인 게임에서는 특정한 이벤트가 있을 때는 캐릭터의 HP, 경험치, 돈을 이벤트 기념으로 주는 경우가 있다. HP와 경험치, 돈의 타입은 다르지만 추가 되는 값은 int 상수로 정해져 있다. 위와 같이 타입은 다르지만 상수를 더 한 값을 얻는 함수를 만들려면 어떻게 해야 할까?

이런 문제도 함수 템플릿으로 해결할 수 있다.

함수 템플릿의 파라미터로 typename만이 아닌 값을 파라미터로 사용할 수도 있다. 아래의 코드는 캐릭터의 HP, 경험치, 돈을 이벤트에서 정해진 값만큼 더해주는 것을 보여준다.

[리스트 2-9]

---

```
#include <iostream>
using namespace std;
```

---

---

```
// 지정된 값만큼 더해준다.
template <typename T, int VAL> T AddValue( T const& CurValue)
{
    return CurValue + VAL;
}

const int EVENT_ADD_HP_VALUE = 50;    // 이벤트에 의해 추가 될 HP 값
const int EVENT_ADD_EXP_VALUE = 30;    // 이벤트에 의해 추가 될 경험치
const int EVENT_ADD_MONEY_VALUE = 10000;    // 이벤트에 의해 추가 될 돈


void main()
{
    int Char_HP = 250;
    cout << Char_HP <<"에서 이벤트에 의해" << AddValue<int,
        EVENT_ADD_HP_VALUE>(Char_HP) << " 로 변경" <<endl;

    float Char_EXP = 378.89f;
    cout << Char_EXP <<"에서 이벤트에 의해" << AddValue<float,
        EVENT_ADD_EXP_VALUE>(Char_EXP) << " 로 변경" <<endl;

    _int64 Char_MONEY = 34567890;
    cout << Char_MONEY <<"에서 이벤트에 의해" << AddValue<_int64,
        EVENT_ADD_MONEY_VALUE>(Char_MONEY) << " 로 변경" <<endl;
}
```

---

실행 결과는 다음과 같다.



```
250에서 이벤트에 의해300로 변경
378.89에서 이벤트에 의해408.89로 변경
34567890에서 이벤트에 의해34577890로 변경
```

앞에서 사용했던 함수 템플릿 사용방법과 조금 달라서 생소할 수도 있을 것이다.

필자가 위에 든 예는 난-타입 함수 템플릿을 사용해야 되는 당위성이 좀 떨어질 수 있다고 생각한다. 하지만 간단한 예제를 사용해서 좀 더 쉽게 설명하기 위한 방법이라 변명해 본다.



난-타입을 사용하는 템플릿은 다음 장에 나오는 클래스 템플릿에서 또 다시 이야기 할 예정이니 잘 기억하고 있기를 바란다. 또, 난-타입을 잘 사용하면 템플릿 메타 프로그래밍을 할 때 큰 도움이 된다. 템플릿 메타 프로그래밍에 대해서는 다음에 설명할 것이다.

### 3 클래스 템플릿

2장에서 함수템플릿에 대해 설명을 했으니 이번에는 클래스 템플릿에 대해서 설명하려고 한다. 클래스 템플릿을 아주 간단하게 말하면 함수 템플릿이 함수에 템플릿을 사용한 것처럼 클래스 템플릿은 클래스에 템플릿을 사용한 것이다.

그러므로 함수 템플릿에 대해서 잘 모른다면 „2장. 함수 템플릿“을 먼저 보고 이 장을 보는 것이 이해하기에 좋다.

#### 3.1 경험치 변경 이력 저장

기획팀에서 유저들이 게임에 접속하여 다른 유저들과 100번의 게임을 했을 때 유저들의 경험치가 변경 되는 이력을 볼 수 있기를 요청했다.

기획팀의 요구를 들어주기 위해서 저는 게임이 끝날 때마다 경험치를 저장한다. 또 경험치 이력 내역을 출력할 때 가장 최신에서 가장 오랜 된 것을 보여줘야 되기 때문에 스택<sup>stack</sup>이라는 자료 구조를 사용한다.

스택은 자료 구조 중의 하나로 가장 마지막에 들어 온 것을 가장 먼저 꺼내는 LIFO(Last In First Out) 형식으로 되어 있다. 데이터를 넣을 때를 push, 빼낼 때는 pop이라는 이름을 일반적으로 사용한다.

경험치 이력을 저장하는 클래스의 구현과 이것을 사용하는 것은 다음과 같다.

[리스트 3-1]

---

```
#include <iostream>

// 경험치를 저장할 수 있는 최대 개수
const int MAX_EXP_COUNT = 100;

// 경험치 저장 스택 클래스
class ExpStack
{
public:
    ExpStack()
    {
```

---

---

```
    Clear();
}

// 초기화 한다.
void Clear()
{
    m_Count = 0;
}

// 스택에 저장된 개수
int Count()
{
    return m_Count;
}

// 저장된 데이터가 없는가?
bool IsEmpty()
{
    return 0 == m_Count ? true : false;
}

// 경험치를 저장한다.
bool push( float Exp )
{
    // 저장할 수 있는 개수를 넘는지 조사한다.
    if( m_Count >= MAX_EXP_COUNT )
    {
        return false;
    }

    // 경험치를 저장 후 개수를 하나 늘린다.
    m_aData[ m_Count ] = Exp;
    ++m_Count;

    return true;
}

// 스택에서 경험치를 빼낸다.
float pop()
```

---

---

```

{
    // 저장된 것이 없다면 0.0f를 반환한다.
    if( m_Count < 1 )
    {
        return 0.0f;
    }

    // 개수를 하나 감소 후 반환한다.
    --m_Count;
    return m_aData[ m_Count ];
}

private:
    float  m_aData[MAX_EXP_COUNT];
    int    m_Count;
};

#include <iostream>
using namespace std;

void main()
{
    ExpStack kExpStack;

    cout << "첫번째 게임 종료- 현재 경험치 145.5f" << endl;
    kExpStack.push( 145.5f );

    cout << "두번째 게임 종료- 현재 경험치 183.25f" << endl;
    kExpStack.push( 183.25f );

    cout << "세번째 게임 종료- 현재 경험치162.3f" << endl;
    kExpStack.push( 162.3f );

    int Count = kExpStack.Count();
    for( int i = 0; i < Count; ++i )
    {
        cout << "현재 경험치->" << kExpStack.pop() << endl;
    }
}

```

---

---

```
}
```

---

[리스트 3-1]의 실행 결과는 다음과 같다.

```
첫번째 게임 종료- 현재 경험치 145.5f
두번째 게임 종료- 현재 경험치 183.25f
세번째 게임 종료- 현재 경험치 162.3f
현재 경험치->162.3
현재 경험치->183.25
현재 경험치->145.5
```

실행 결과를 보면 알 수 있듯이 스택 자료구조를 사용하였기 때문에 제일 뒤에 넣은 데이터가 가장 먼저 출력되었다.

## 3.2 게임 돈 변경 이력도 저장해 주세요

경험치 변경 이력을 저장하고 출력하는 기능을 만들어서 기획팀에 보여주기 이번에는 게임 돈의변경 이력도 보고 싶다고 말한다.

위에서 경험치 변경 이력 저장 기능을 만들어 보았으니 금방 할 수 있는 것이다. 그래서 이번에는 이전 보다 훨씬 더 빨리 만들었다.

### [리스트 3-2]

---

```
#include <iostream>

// 돈을 저장할 수 있는 최대 개수
const int MAX_MONEY_COUNT = 100;

// 돈 저장 스택 클래스
class MoneyStack
{
public:
    MoneyStack()
    {
        Clear();
    }
}
```

---

---

```
// 초기화 한다.
void Clear()
{
    m_Count = 0;
}

// 스택에 저장된 개수
int Count()
{
    return m_Count;
}

// 저장된 데이터가 없는가?
bool IsEmpty()
{
    return 0 == m_Count ? true : false;
}

// 돈을 저장한다.
bool push( __int64 Money )
{
    // 저장 할 수 있는 개수를 넘는지 조사한다.
    if( m_Count >= MAX_MONEY_COUNT )
    {
        return false;
    }

    // 저장후 개수를 하나 늘린다.
    m_aData[ m_Count ] = Money;
    ++m_Count;

    return true;
}

// 스택에서 돈을 빼낸다.
__int64 pop()
{
    // 저장된 것이 없다면 0을 반환한다.
```

---

---

```

    if( m_Count < 1 )
    {
        return 0;
    }

    // 개수를 하나 감소 후 반환한다.
    --m_Count;
    return m_aData[ m_Count ];
}

private:
    __int64 m_aData[MAX_MONEY_COUNT];
    int m_Count;
};

```

---

ExpStack 클래스와 MoneyStack 클래스가 비슷하다

게임 돈 변경 이력 저장 기능을 가지고 있는 MoneyStack 클래스를 만들고 보니 앞에 만든 ExpStack와 거의 같다. 저장하는 데이터의 자료형만 다를 뿐이지 모든 것이 같다. 그리고 기획팀에서는 게임 캐릭터의 Level 변경 이력도 저장하여 보여주기를 바라는 것 같다. 이미 거의 똑같은 클래스를 두 개 만들었고 앞으로도 기획팀에서 요청이 있으면 더 만들 것 같다. 이렇게 자료형만 다른 클래스를 어떻게 하면 하나의 클래스로 정의 할 수 있을까? 이와 비슷한 문제를 이전의 "함수 템플릿"에서도 나타나지 않았나? 그때 어떻게 해결할 수 있을까?(생각나지 않으면 앞 2장의 "함수 템플릿"을 다시 한번 확인한다)

템플릿으로 하면 된다. 기능은 같지만 변수의 자료형만 다른 함수를 템플릿을 사용하여 하나의 함수로 정의했듯이 이번에는 템플릿을 사용하여 클래스를 정의한다. 클래스에서 템플릿을 사용하면 이것을 클래스 템플릿이라고 한다. 클래스 템플릿을 사용하면 위에서 중복된 클래스를 하나의 클래스로 만들 수 있다.

### 3.3 클래스 템플릿을 사용하는 방법

클래스 템플릿을 정의하는 문법은 다음과 같다.

```
template <typename type> class 클래스 이름
{
    .....
};
```

```
template <typename T> class Stack
{
    .....
};
```

정의한 클래스 템플릿을 사용하는 방법은 다음과 같다.

Class name <type> 변수명;

Stack <int> kStack;

### 3.4 Stack 템플릿 클래스

지금까지 만들었던 ExpStack 과 MoneyStack을 클래스 템플릿으로 만든 코드는 다음과 같다.

[리스트 3-3]

---

```
#include <iostream>

const int MAX_COUNT = 100;

template<typename T>
class Stack
{
```

---



---

```
public:
    Stack()
    {
        Clear();
    }

    // 초기화 한다.
    void Clear()
    {
        m_Count = 0;
    }

    // 스택에 저장된 개수
    int Count()
    {
        return m_Count;
    }

    // 저장된 데이터가 없는가?
    bool IsEmpty()
    {
        return 0 == m_Count ? true : false;
    }

    // 데이터를 저장한다.
    bool push( T data )
    {
        // 저장 할수 있는 개수를 넘는지 조사한다.
        if( m_Count >= MAX_COUNT )
        {
            return false;
        }

        // 저장후 개수를 하나 늘린다.
        m_aData[ m_Count ] = data;
        ++m_Count;

        return true;
    }
}
```

---

---

```

// 스택에서 빼낸다.
T pop()
{
    // 저장된 것이 없다면 0을 반환한다.
    if( m_Count < 1 )
    {
        return 0;
    }

    // 개수를 하나 감소 후 반환한다.
    --m_Count;
    return m_aData[ m_Count ];
}

private:
    T m_aData[MAX_COUNT];
    int m_Count;
};

#include <iostream>
using namespace std;

void main()
{
    Stack<double> kStackExp;

    cout << "첫번째 게임 종료- 현재 경험치 145.5f" << endl;
    kStackExp.push( 145.5f );

    cout << "두번째 게임 종료- 현재 경험치 183.25f" << endl;
    kStackExp.push( 183.25f );

    cout << "세번째 게임 종료- 현재 경험치 162.3f" << endl;
    kStackExp.push( 162.3f );

    int Count = kStackExp.Count();
    for( int i = 0; i < Count; ++i )

```

---

---

```
{
    cout << "현재 경험치->" << kStackExp.pop() << endl;
}

cout << endl << endl;

Stack<__int64> kStackMoney;

cout << "첫번째 게임 종료- 현재 돈 1000023" << endl;
kStackMoney.push( 1000023 );

cout << "두번째 게임 종료- 현재 돈 1000234" << endl;
kStackMoney.push( 1000234 );

cout << "세번째 게임 종료- 현재 돈 1000145" << endl;
kStackMoney.push( 1000145 );

Count = kStackMoney.Count();
for( int i = 0; i < Count; ++i )
{
    cout << "현재 돈->" << kStackMoney.pop() << endl;
}
}
```

---

[리스트 3-3]의 실행 결과는 다음과 같다.

```
첫번째 게임 종료- 현재 경험치 145.5f
두번째 게임 종료- 현재 경험치 183.25f
세번째 게임 종료- 현재 경험치 162.3f
현재 경험치->162.3
현재 경험치->183.25
현재 경험치->145.5

첫번째 게임 종료- 현재 돈 1000023
두번째 게임 종료- 현재 돈 1000234
세번째 게임 종료- 현재 돈 1000145
현재 돈->1000145
현재 돈->1000234
현재 돈->1000023
```

클래스 템플릿으로 Stack을 구현하여 앞으로 다양한 데이터를 사용할 수 있게 되었다. 그런데 위의 Stack 클래스는 부족한 부분이 있다. 앞으로 이 부족한 부분을 채워 나가면서 클래스 템플릿에 대해서 좀 더 알아보자.

### 3.5 클래스 템플릿에서 non-type 파라미터 사용

위에서 만든 Stack 클래스는 데이터를 저장할 수 있는 공간이 100개로 정해져 있다. Stack의 크기는 사용하는 곳에 따라서 변할 수 있어야 사용하기에 적합하다.

함수 템플릿을 설명할 때도 non-type이 나왔는데 사용방법이 거의 같다. 템플릿 파라미터를 기본 데이터 형으로 한다. 다음 사용 예를 참고한다.

#### [리스트 3-4]

```
// 템플릿 파라미터중 int Size가 non-type 파라미터이다.
template <typename T, int Size>
class Stack
{
public:
    Stack()
    {
        Clear();
    }
}
```

---

```
// 초기화 한다.
void Clear()
{
    m_Count = 0;
}

// 스택에 저장된 개수
int Count()
{
    return m_Count;
}

// 저장된 데이터가 없는가?
bool IsEmpty()
{
    return 0 == m_Count ? true : false;
}

// 데이터를 담을수 있는 최대 개수
int GetStackSize()
{
    return Size;
}

// 데이터를 저장한다.
bool push( T data )
{
    // 저장할 수 있는 개수를 넘는지 조사한다.
    if( m_Count >= Size )
    {
        return false;
    }

    // 저장 후 개수를 하나 늘린다.
    m_aData[ m_Count ] = data;
    ++m_Count;

    return true;
}
```

---

---

```

    }

    // 스택에서 빼낸다.
    T pop()
    {
        // 저장된 것이 없다면 0을 반환한다.
        if( m_Count < 1 )
        {
            return 0;
        }

        // 개수를 하나 감소 후 반환한다.
        --m_Count;
        return m_aData[ m_Count ];
    }

private:
    T m_aData[Size];
    int m_Count;
};

#include <iostream>
using namespace std;

void main()
{
    Stack<int, 100> kStack1;
    cout << "스택의 크기는?" << kStack1.GetStackSize() << endl;

    Stack<double, 60> kStack2;
    cout << "스택의 크기는?" << kStack2.GetStackSize() << endl;
}

```

---

[리스트 3-4]의 실행 결과는 다음과 같다.

```
스택의 크기는?100
스택의 크기는?60
```

### 3.6 템플릿 파라미터 디폴트 값 사용

일반 함수에서 함수 인자의 디폴트 값을 지정하듯이 클래스 템플릿의 파라미터도 디폴트 값으로 할 수 있다.

[리스트 3-5]

---

```
// 템플릿 파라미터 중 int Size가 non-type 파라미터다.
// Size의 디폴트 값을 100으로 한다.
#include <iostream>

template<typename T, int Size=100>
class Stack
{
    ..... 생략
}

void main()
{
    Stack<int, 64> kStack1;
    cout << "스택의 크기는?" << kStack1.GetStackSize() << endl;

    Stack<double> kStack2;
    cout << "스택의 크기는?" << kStack2.GetStackSize() << endl;
}
```

---

[리스트 3-5]에서 템플릿 파라미터 중 Size의 값을 디폴트 100으로 한다. 클래스를 생성할 때 두 번째 파라미터 값을 지정하지 않으면 디폴트 값을 사용한다.

[리스트 3-5]의 실행 결과는 다음과 같다.

```
스택의 크기는?64
스택의 크기는?100
```

### 3.7 스택 클래스의 크기를 클래스 생성자에서 지정

클래스 템플릿에 대한 설명을 계속 하기 위해 현재까지 만든 스택 클래스를 변경한다. 스택의 크기를 클래스 템플릿 파라미터가 아닌 생성자에서 지정하도록 변경할 것이다.

[리스트 3-6]

---

```
template<typename T, int Size=100> class Stack
{
public:
    explicit Stack( int size )
    {
        m_Size = size;
        m_aData = new T[m_Size];

        Clear();
    }

    ~Stack()
    {
        delete[] m_aData;
    }

    // 초기화 한다.
    void Clear()
    {
        m_Count = 0;
    }

    // 스택에 저장된 개수
    int Count()
```

---



---

```
{
    return m_Count;
}

// 저장된 데이터가 없는가?
bool IsEmpty()
{
    return 0 == m_Count ? true : false;
}

// 데이터를 담을 수 있는 최대 개수
int GetStackSize()
{
    return m_Size;
}

// 데이터를 저장한다.
bool push( T data )
{
    // 저장할 수 있는 개수를 넘는지 조사한다.
    if( m_Count >= m_Size )
    {
        return false;
    }

    // 저장 후 개수를 하나 늘린다.
    m_aData[ m_Count ] = data;
    ++m_Count;

    return true;
}

// 스택에서 빼낸다.
T pop()
{
    // 저장된 것이 없다면 0을 반환한다.
    if( m_Count < 1 )
    {
        return 0;
    }
}
```

---

---

```

    }

    // 개수를 하나 감소 후 반환한다.
    --m_Count;
    return m_aData[ m_Count ];
}

private:
    T* m_aData;
    int m_Count;

    int m_Size;
};

#include <iostream>
using namespace std;

void main()
{
    Stack<int> kStack1(64);
    cout << "스택의 크기는? " << kStack1.GetStackSize() << endl;
}

```

---

[리스트 3-6]의 실행 결과는 다음과 같다.

```
스택의 크기는? 64
```

[리스트 3-6]의 코드에서 잘 보지 못한 키워드가 있을 것이다. 바로 `explicit` 이다. `explicit` 키워드로 규정된 생성자는 암시적인 형 변환을 할 수 없다. 그래서 [리스트 3-6]의 `void main()`에서 다음과 같이 클래스를 생성하면 컴파일 에러가 발생합니다.

```
Stack kStack1 = 64;
```

## 3.8 클래스 템플릿 전문화

기획팀에서 새로운 요구가 들어왔다. 이번에는 게임을 할 때 같이 게임을 했던 유저의 아이디를 저장하여 보여주기를 원한다. 지금까지 만든 Stack 클래스는 기본 자료형을 사용하는 것을 전제로 했는데 유저의 아이디를 저장하려면 문자열이 저장되어야 하므로 사용할 수가 없다.

기본 자료형으로 하지 않고 문자열을 사용한다는 것만 다르지 작동은 비슷하므로 Stack이라는 이름의 클래스를 사용하고 싶다. 기존의 Stack 클래스 템플릿과 클래스의 이름만 같지 행동은 다른 Stack 클래스를 구현 하려고 한다. 이때 필요한 것인 클래스 템플릿의 전문화라는 것이다. 클래스 템플릿 전문화는 기존에 구현한 클래스 템플릿과 비교해서 이름과 파라미터 개수는 같지만 파라미터를 특정한 것으로 지정합니다.

전문화된 클래스 템플릿 정의는 다음과 같은 형태를 가진다.

---

```
template < class 클래스 이름<지정된 타입>
{
    .....
};
```

---

### [리스트 3-7]

---

```
// ID 문자열의 최대 길이(null 문자포함)
const int MAX_ID_LENGTH = 21;
```

```
template<typename T> class Stack
{
public:
    explicit Stack( int size )
    {
        m_Size = size;
        m_aData = new T[m_Size];

        Clear();
    }
```

---

---

```
~Stack()
{
    delete[] m_aData;
}

// 초기화 한다.
void Clear()
{
    m_Count = 0;
}

// 스택에 저장된 개수
int Count()
{
    return m_Count;
}

// 저장된 데이터가 없는가?
bool IsEmpty()
{
    return 0 == m_Count ? true : false;
}

// 데이터를 담을 수 있는 최대 개수
int GetStackSize()
{
    return m_Size;
}

// 데이터를 저장한다.
bool push( T data )
{
    // 저장할 수 있는 개수를 넘는지 조사한다.
    if( m_Count >= m_Size )
    {
        return false;
    }

    // 저장 후 개수를 하나 늘린다.
```

---

---

```

    m_aData[ m_Count ] = data;
    ++m_Count;

    return true;
}

// 스택에서 빼낸다.
T pop()
{
    // 저장된 것이 없다면 0을 반환한다.
    if( m_Count < 1 )
    {
        return 0;
    }

    // 개수를 하나 감소 후 반환한다.
    --m_Count;
    return m_aData[ m_Count ];
}

private:
    T* m_aData;
    int m_Count;

    int m_Size;
};

#include <string>
// 아래의 코드는 문자열을 저장하기 위해 char* 으로 전문화한 Stack 클래스입니다.
template<T> class Stack<char*>
{
public:
    explicit Stack( int size )
    {
        m_Size = size;

        m_ppData = new char *[m_Size];
        for( int i = 0; i < m_Size; ++i )
        {

```

---

---

```

        m_ppData[i] = new char[MAX_ID_LENGTH];
    }

    Clear();
}

~Stack()
{
    for( int i = 0; i < m_Size; ++i )
    {
        delete[] m_ppData[i];
    }

    delete[] m_ppData;
}

// 초기화한다.
void Clear()
{
    m_Count = 0;
}

// 스택에 저장된 개수
int Count()
{
    return m_Count;
}

// 저장된 데이터가 없는가?
bool IsEmpty()
{
    return 0 == m_Count ? true : false;
}

// 데이터를 담을 수 있는 최대 개수
int GetStackSize()
{
    return m_Size;
}

```

---

---

```

// 데이터를 저장한다.
bool push( char* pID )
{
    // 저장할 수 있는 개수를 넘는지 조사한다.
    if( m_Count >= m_Size )
    {
        return false;
    }

    // 저장 후 개수를 하나 늘린다.
    strncpy_s( m_ppData[m_Count], MAX_ID_LENGTH, pID, MAX_ID_LENGTH - 1);
    m_ppData[m_Count][MAX_ID_LENGTH - 1] = '\0';

    ++m_Count;

    return true;
}

// 스택에서 빼낸다.
char* pop()
{
    // 저장된 것이 없다면 0을 반환한다.
    if( m_Count < 1 )
    {
        return 0;
    }

    // 개수를 하나 감소 후 반환한다.
    --m_Count;
    return m_ppData[ m_Count ];
}

private:
    char** m_ppData;
    int m_Count;

    int m_Size;
};

```

---

---

```
#include <iostream>
using namespace std;

void main()
{
    Stack<int> kStack1(64);
    cout << "스택의 크기는? " << kStack1.GetStackSize() << endl;
    kStack1.push(10);
    kStack1.push(11);
    kStack1.push(12);

    int Count1 = kStack1.Count();
    for( int i = 0; i < Count1; ++i )
    {
        cout << "유저의 레벨 변화 -> " << kStack1.pop() << endl;
    }

    cout << endl;

    char GameID1[MAX_ID_LENGTH] = "NiceChoi";
    char GameID2[MAX_ID_LENGTH] = "SuperMan";
    char GameID3[MAX_ID_LENGTH] = "Attom";

    // Stack 클래스 템플릿의 char* 전문화 버전을 생성한다.
    Stack<char*> kStack2(64);
    kStack2.push(GameID1);
    kStack2.push(GameID2);
    kStack2.push(GameID3);

    int Count2 = kStack2.Count();
    for(int i = 0; i < Count2; ++i)
    {
        cout << "같이 게임을 한유저의 ID -> " << kStack2.pop() << endl;
    }
}
```

---



[리스트 3-7]의 실행 결과는 다음과 같다.

```
스택의 크기는? 64
유저의 레벨 변화 -> 12
유저의 레벨 변화 -> 11
유저의 레벨 변화 -> 10

같이 게임을 한 유저의 ID -> Atton
같이 게임을 한 유저의 ID -> SuperMan
같이 게임을 한 유저의 ID -> NiceChoi
```

### 3.9 클래스 템플릿 부분 전문화

클래스 템플릿은 템플릿 파라미터 중 일부를 구체적인 형(type)을 사용, 또는 템플릿 파라미터를 포인터나 참조를 사용하여 부분 전문화를 할 수 있다.

구체적인 형 사용에 의한 부분 전문화

```
template< typename T1, typename T2 > class Test { .... };
```

의 T2를 float로 구체화 하여 부분 전문화를 하면 다음과 같다.

```
template< typename T1 > class Test { ..... };
```

코드는 다음과 같다.

[리스트 3-8]

---

```
template< typename T1, typename T2 >
class Test
{
public:
    T1 Add( T1 a, T2 b )
    {
        cout << "일반 템플릿을 사용했습니다." << endl;
        return a;
    }
};
```

---

---

```
// T2를 float로 구체화한 Test의 부분 전문화 템플릿
template< typename T1 >
class Test<T1,float>
{
public:
    T1 Add( T1 a, float b )
    {
        cout << "부분 전문화 템플릿을 사용했습니다." << endl;
        return a;
    }
};

#include <iostream>
using namespace std;

void main()
{
    Test<int, int> test1;
    test1.Add( 2, 3 );

    Test<int, float> test2;
    test2.Add( 2, 5.8f );
}
```

---

위의 예에서는 템플릿 파라미터 2개 중 일부를 구체화하여 부분 전문화를 했지만, 2개 이상도 가능하다.

```
template< typename T1, typename T2, typename T3 > class Test { .... };
```

의 부분 전문화 템플릿은

```
template< typename T1, typename T2 > class Test { ..... };
```

- 포인터의 부분 전문화

```
template< typename T > class TestP { .... };
```

의 T의 T\* 부분 전문화를 하는 다음과 같다.

```
template< typename T > class TestP { ..... };
```

코드는 다음과 같다.

### [리스트 3-9]

---

```
template< typename T >
class TestP
{
public:
    void Add()
    {
        cout << "일반 템플릿을 사용했습니다." << endl;
    }
};

// T를 T*로 부분 전문화
template< typename T >
class TestP<T*>
{
public:
    void Add()
    {
        cout << "포인터를 사용한 부분 전문화 템플릿을 사용했습니다." << endl;
    }
};

#include <iostream>
using namespace std;

void main()
{
    TestP<int> test1;
    test1.Add();

    TestP<int*> test2;
    test2.Add();
}
```

---

[리스트 3-9]의 실행 결과는 다음과 같다.

```
일반 템플릿을 사용했습니다.  
포인터를 사용한 부분 전문화 템플릿을 사용했습니다.
```

### 3.10 싱글톤 템플릿 클래스

클래스 상속을 할 때 템플릿 클래스를 상속 받음으로 상속 받는 클래스의 기능을 확장할 수 있다. 필자의 경우 현업에서 클래스 템플릿을 가장 많이 사용하는 경우가 클래스 템플릿을 사용한 싱글톤 클래스 템플릿을 사용하는 것이다.

어떠한 객체가 꼭 하나만 있어야 되는 경우 싱글톤으로 정의한 클래스 템플릿을 상속 받도록 한다.

싱글톤은 싱글톤 패턴을 말하는 것으로 어떤 클래스의 인스턴스가 꼭 하나만 생성되도록 하며, 전역적인 접근이 가능하도록 한다. 어떤 클래스를 전역으로 사용하는 경우 복수개의 인스턴스가 생성되지 않도록 싱글톤 패턴으로 생성하는 것을 권장한다.

사용하는 방법은 베이스 클래스를 템플릿을 사용하여 만든다. 그리고 이것을 상속 받는 클래스에서 베이스 클래스의 템플릿 파라미터에 해당 클래스를 사용한다. 즉 싱글톤 클래스 템플릿은 이것을 상속 받는 클래스를 싱글톤으로 만들어 준다.

위에서 설명한 클래스 템플릿에 대하여 이해를 했다면 다음의 코드를 보면 더 잘 이해를 할 수 있으리라 생각한다. 싱글톤 클래스 템플릿은 직접 생성을 하지 않으므로 주 멤버들을 static로 만들어준다. 그리고 생성자를 통해서 \_Singleton를 생성하지 않고 GetSingleton()을 통해서만 생성하도록 한다.

#### [리스트 3-10]

```
// 파라미터 T를 싱글톤이 되도록 정의한다.  
#include <iostream>  
using namespace std;  
  
template <typename T>
```

---

```

class MySingleton
{
public:
    MySingleton() {}
    virtual ~MySingleton() {}

    // 이 멤버를 통해서만 생성이 가능하다.
    static T* GetSingleton()
    {
        // 아직 생성이 되어 있지 않으면 생성한다.
        if( NULL == _Singleton ) {
            _Singleton = new T;
        }

        return ( _Singleton );
    }

    static void Release()
    {
        delete _Singleton;
        _Singleton = NULL;
    }

private:
    static T* _Singleton;
};

template <typename T> T* MySingleton<T>::_Singleton = NULL;

// 싱글톤 클래스 템플릿을 상속 받으면서 파라미터에 본 클래스를 넘긴다.
class MyObject : public MySingleton<MyObject>
{
public:
    MyObject() : _nValue(10) {}

    void SetValue( int Value ) { _nValue = Value;}
    int GetValue() { return _nValue; }

private :

```

---

---

```

int _nValue;
};

void main()
{
    MyObject* MyObj1 = MyObject::GetSingleton();

    cout << MyObj1->GetValue() << endl;

    // MyObj2는 MyObj1과 동일한 객체이다.
    MyObject* MyObj2 = MyObject::GetSingleton();
    MyObj2->SetValue(20);

    cout << MyObj1->GetValue() << endl;
    cout << MyObj2->GetValue() << endl;
}

```

---

[리스트 3-10]의 실행 결과는 다음과 같다.



```

10
20
20

```

### 3.11 클래스 템플릿 코딩 스타일 개선

위에서 예제로 구현한 다양한 클래스 템플릿의 코딩 스타일은 클래스 선언 안에서 각 멤버들의 정의를 구현하고 있다. 클래스의 코드 길이가 크지 않은 경우는 코드를 보는데 불편하지 않지만 코드 길이가 길어지는 경우 클래스의 전체적인 윤곽을 바로 알아보기가 쉽지 않다.

긴 코드를 가지는 클래스 템플릿의 경우는 클래스의 선언과 정의를 분리하는 것이 좋다. 위에서 예제로 나온 클래스 템플릿 중 의 Stack 클래스 템플릿을 선언과 정의를 분리하면 다음과 같다.

[리스트 3-11]

---

```

#include <iostream>

```

---

---

```
using namespace std;

template<typename T> class Stack
{
public:
    explicit Stack( int size );

    ~Stack();

    // 초기화 한다.
    void Clear();

    // 스택에 저장된 개수
    int Count();

    // 저장된 데이터가 없는가?
    bool IsEmpty();

    // 데이터를 담을 수 있는 최대 개수
    int GetStackSize();

    // 데이터를 저장한다.
    bool push( T data );

    // 스택에서 빼낸다.
    T pop();

private:
    T* m_aData;
    int m_Count;

    int m_Size;
};

template < typename T > Stack<T>::Stack( int size )
{
    m_Size = size;
    m_aData = new T[m_Size];
```

---

---

```
    Clear();
}

template < typename T > Stack<T>::~~Stack()
{
    delete[] m_aData;
}

template < typename T > void Stack<T>::Clear()
{
    m_Count = 0;
}

template < typename T > int Stack<T>::Count()
{
    return m_Count;
}

template < typename T > bool Stack<T>::IsEmpty()
{
    return 0 == m_Count ? true : false;
}

template < typename T > int Stack<T>::GetStackSize()
{
    return m_Size;
}

template < typename T > bool Stack<T>::push(T data )
{
    // 저장할 수 있는 개수를 넘는지 조사한다.
    if( m_Count >= m_Size )
    {
        return false;
    }

    // 저장 후 개수를 하나 늘린다.
    m_aData[ m_Count ] = data;
    ++m_Count;
}
```

---



---

```

    return true;
}

template < typename T > T Stack<T>::pop()
{
    // 저장된 것이 없다면 0을 반환한다.
    if( m_Count < 1 )
    {
        return 0;
    }

    // 개수를 하나 감소 후 반환한다.
    --m_Count;
    return m_aData[ m_Count ];
}

void main()
{
    Stack<int> kStack1(7);
    cout << "스택의 크기는?" << kStack1.GetStackSize() << endl;
}

```

---

[리스트 3-11]의 코드를 보면 알 수 있듯이 클래스 안에 정의를 했던 것과의 차이점은 클래스 멤버 정의를 할 때 템플릿 선언하고 클래스 이름에 템플릿 파라미터를 적어 준다.

### 3.12 클래스 선언과 정의를 각각 다른 파일에 하려면

일반적인 클래스의 경우 크기가 작은 경우를 제외하면 클래스의 선언과 정의를 서로 다른 파일에 한다.

클래스 템플릿의 경우는 일반적인 방법으로는 그렇게 할 수 없다. 클래스 멤버 정의를 선언과 다른 파일에 하려면 멤버 정의를 할 때 'export'라는 키워드를 사용한다. [리스트 3-11]의 GetStackSize()에 export를 사용하면 다음과 같이 된다.

---

```
template < typename T >
export int Stack::GetStackSize()
{
    return m_Size;
}
```

---

그러나 export라는 키워드를 사용하면 컴파일 에러가 발생한다. 이유는 현재 대부분의 C++ 컴파일러에서는 export라는 키워드를 지원하지 않는다. 왜냐하면, 이것을 지원하기 위해 필요로 하는 노력은 컴파일러를 새로 만들 정도의 노력을 필요로 할 정도로 어렵다고 한다. 현재까지도 대부분의 컴파일러 개발자들은 구현 계획을 세우지도 않고 있으며 일부에서는 구현에 반대하는 의견도 있다고 한다.

그럼 클래스 템플릿의 선언과 정의를 서로 다른 파일에 할 수 있는 방법은 없을까? 약간 편법을 사용하면 가능하다.

inline이라는 의미를 가지고 있는 '.inl' 확장자 파일에 클래스 구현하고 이 .inl 파일을 헤더 파일에서 포함한다. (참고로 .inl 파일을 사용하는 것은 일반적인 방식은 아니고 일부 라이브러리나 상용 3D 엔진에서 간혹 사용하는 것을 볼 수 있다).

[리스트 3-11]의 Stack 클래스 템플릿의 선언과 정의를 다른 파일로 하는 예의 일부를 다음에 구현했다.

#### [리스트 3-12]

---

```
// stack.h 파일
template<typename T>
class Stack
{
public:

    void Clear();

};

#include <iostream>
```

---

---

```
#include "stack.hpp"

// stack.inl 파일

template < typename T >
void Stack<T>::Clear()
{
    std::cout << "Clear() " << std::endl;
}

// stack.cpp 파일
#include "stack.h"

void main()
{
    Stack<int> k;
    k.Clear();
}
```

---

이것으로 클래스 템플릿에 대한 설명을 마쳤다. 함수 템플릿에 대한 글을 봤다면 템플릿에 대한 어느 정도 이해를 가지고 있을 테니 어렵지 않게 이해를 할 수 있으리라 생각하지만 필자의 부족한 글 때문에 이해가 어렵지 않았을까라는 걱정도 조금한다.

글만 보고 넘기지 말고 직접 코딩 해 보기를 권장한다. 본문에 나오는 예제들은 모두 코드 길이가 짧은 것이어서 직접 코딩을 하더라도 긴 시간은 걸리지 않을 것이다.

다음 장부터는 본격적으로 STL에 대한 설명에 들어간다. 앞에서 이야기 했듯 STL은 템플릿으로 만들어진 것이다. 아직 템플릿의 유용성을 느끼지 못한 독자들은 STL에 대해서 알게 되면 템플릿의 뛰어남을 알게 되리라 생각한다.

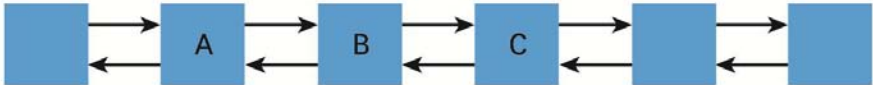
## 4 연결 리스트

이번 장부터는 본격적으로 STL에 대해서 이야기한다. STL은 C++ 템플릿을 사용해 만든 표준 라이브러리이다. 그러므로 템플릿에 대해서 아직 잘 모르시는 독자라면 앞 장에 있는 템플릿에 대한 부분을 먼저 읽어보기를 권한다. 일반적으로 STL 중에서 가장 많이 사용하는 라이브러리는 컨테이너 라이브러리이다. 컨테이너는 말 그대로 무엇인가를 담는 것이다. 컨테이너는 int나 float 등의 기본 자료 형이나 구조체, 클래스 같은 사용자 정의형을 담는다. STL의 컨테이너는 list, vector, deque, map, set이 있다. 이번 장에서는 list에 대해서 이야기한다.

### 4.1 list의 자료구조

list는 자료구조 중 '연결 리스트'를 템플릿으로 구현한 것이다. 그래서 list를 알려면 '연결 리스트'라는 자료구조의 이해가 꼭 필요하다. 연결 리스트는 단어 그 자체로 해석하면 "(무엇인가)서로 연결 되어 줄지어 있다"라고 말할 수 있다. 말보다는 그림을 보는 것이 이해하기 쉬울 테니 그림 4-1을 보자.

그림 4-1 연결 리스트



### 4.2 연결 리스트의 특징

1. 고정 길이인 배열에 비해 길이가 가변적이다.  
배열은 처음에 설정한 크기 이외에는 더 이상 데이터를 담을 수 없지만 연결 리스트는 동적으로 크기를 변경 할 수 있다.
2. 중간에 데이터 삽입, 삭제가 용이하다.  
데이터를 중간에 삽입할 때 배열은 그림 4-1에서 B와 C 사이에 새로운 데이터를 넣는다면 그림 4-2와 같이 C 이후의 데이터를 모두 뒤로 이동 해야 한다. 그러나 연결 리스트는 그림 4-3과 같이 B와 C 사이에 넣으면서 연결

고리만 바꾸면 됩니다.

그림 4-2 배열에서 데이터 삽입하기

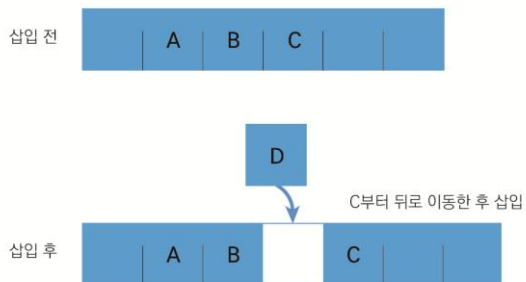


그림 4-3 연결 리스트에서 데이터 삽입하기

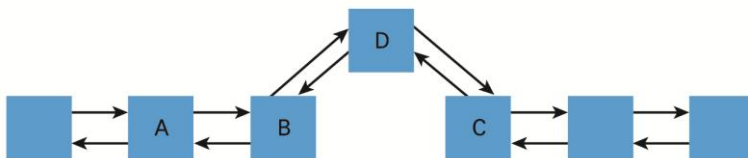
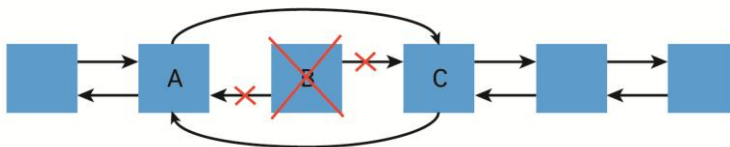


그림 4-2의 B를 삭제 하면 배열은 C 이후의 모든 데이터를 앞으로 이동해야 한다. 그러나 연결 리스트는 그림 4-4와 같이 B를 삭제하고 B의 연결 고리를 없애면 됩니다.

그림 4-4 리스트에서 데이터 삭제하기



이렇게 연결 리스트는 배열에 비해서 크기가 가변적이고, 중간에 데이터 삭제와 삽입이 용이하다는 장점이 있다. 그렇지만 단점으로는 배열에 비해서 데이터의 삽입과 삭제를 구현하기 어렵고 내부 절차가 복잡하다. 배열은 랜덤하게 접근할 수 있지만 연결 리스트는 랜덤하게 접근할 수 없다. 연결 리스트는 특징을 잘 파악한 후 알맞은 곳에 사용해야 한다.

### 4.3 STL list를 사용하면 좋은 점

STL을 사용하지 않는다면 C/C++ 언어, 자료구조를 공부하고 필요한 자료구조를 직접 만들어 사용해야 한다. 직접 만들어 사용하면 여러 번 되풀이(프로젝트나 회사가 바뀌면)하여 만들어야 하므로 불필요한 시간을 소비하고, 연결 리스트 자료구조를 잘못 구현하여 버그를 만들 위험이 있고, 개인마다 구현 방법이 다르므로 사용이나 유지보수 측면에서 불편하다.

그러나 STL list(이하 list)를 사용하면 연결 리스트를 따로 만들어야 하는 시간을 절약할 수 있고, 이미 검증되어 있으므로 안전하고, 표준 라이브러리이므로 사용 방법이 언제나 같아서 사용 및 유지보수가 좋아진다.

다만, list를 사용할 때는 특성을 잘 파악하여 올바르게 사용해야 한다. list를 적합하지 않은 곳에 사용하면 성능의 하락 및 시스템 에러를 유발할 위험이 생긴다.

#### 4.3.1 STL에 버그가 있다?

현업에 일할 때 STL을 안 쓸 수도 있다. STL을 사용하지 않는 이유가 STL을 사용했을 때 잘 알 수 없는 문제가 발생했는데 STL을 사용하지 않으니 괜찮아졌다는 이유로 STL에 버그가 있다고 생각하는 경우가 있다. 필자가 생각하기에는 STL의 버그가 아닌 다른 곳에서 발생한 문제이든가 STL의 특징을 제대로 파악하지 못하고 사용해서 일어난 문제라고 생각한다. 만약 정말 STL에 버그가 있다면 이런 중요한 문제는 널리 알려져서 다른 프로그래머들에게 도움을 주고 큰 버그를 찾은 스타(?)가 되는 것은 어떨까?

### 4.3.2 list를 사용해야 하는 경우

#### 1. 저장할 데이터 개수가 가변적이다.

저장할 데이터 개수가 정해져 있지 않은 경우 배열은 설정된 크기를 넘어가면 데이터가 넘쳐서 실행 도중 프로그램 오류가 발생하므로 코드를 수정 후 재컴파일해야 한다. 그렇다고 배열에 설정된 크기가 변할 때마다 재컴파일하는 것을 방지하려고 넉넉한 크기로 큰 배열을 만든다면 메모리 낭비가 발생한다. 그러나 list를 사용하면 저장 공간의 크기가 자동으로 변하므로 유연하게 사용할 수 있다.

대형 프로그램을 만들어 본 적이 없는 분들은 컴파일에 걸리는 시간은 짧은 시간이라고 생각하여 컴파일을 자주 하는 것에 대한 문제를 느끼지 못할 수도 있다. 그러나 일반적으로 콘솔이나 PC 게임은 클라이언트 프로그램을 컴파일 하는데 걸리는 시간은 15 ~ 30분 이상 걸리는 경우가 많다.

#### 2. 중간에 데이터 삽입이나 삭제가 자주 일어난다.

MMORPG 게임은 지도가 아주 크고 게임상에서 어떤 캐릭터의 행동에 대한 정보를 근처의 클라이언트에게만 통보하므로 지도를 작은 단위로(보통 사각형으로) 나눈 후 같은 단위에 포함 되어 있는 클라이언트와 그 단위 근처의 클라이언트에게만 통보한다. 지도를 작은 단위로 분할하여 해당 영역에 들어오는 유저는 저장하고 나가는 유저는 삭제를 해야 한다. 이와 같이 빈번하게 삽입과 삭제가 일어나는 곳에 list를 사용한다.

그림 4-5 접속한 클라이언트 간의 인접 위치를 나타낸 지도

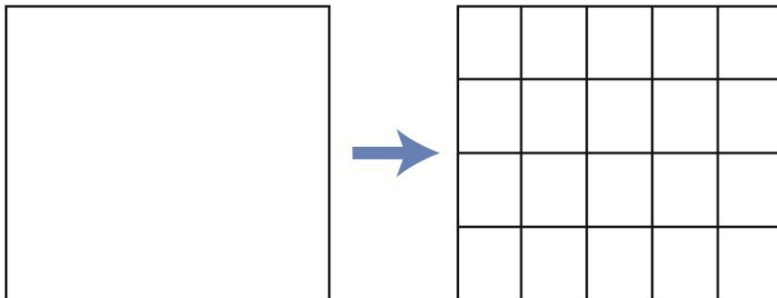


그림 4-5 하나의 지도로 접속한 클라이언트간의 인접 위치를 관리하는 것은 너무 비효율적이므로 오른쪽과 같이 지도를 작은 단위로 나눈 후 접속한 클라이언트를 단위 별로 관리한다

3. 저장할 데이터 개수가 많으면서 검색을 자주 한다면 다른 컨테이너 라이브러리를 사용해야 한다.

아주 많은 데이터를 저장하면서 특정 데이터를 자주 검색해야 할 때 list를 사용하면 검색 속도가 많이 느려지므로 이런 경우에는 map이나 set, hash\_map을 사용해야 한다.

4. 데이터를 랜덤하게 접근하는 경우가 많지 않다.

배열은 랜덤 접근이 가능하나 list는 순차 접근만 가능하다. 그래서 저장된 위치를 알더라도 반복자(iterator)(아래에 설명하겠다)를 통해서 접근해야 한다. 아이템을 자주 사용하는 온라인 게임에서는 아이템 사용 시 아이템 정보에 빈번하게 접근하므로 성능을 위해 메모리 낭비를 감수하고 배열로 데이터를 저장해서 랜덤 접근을 사용하게 한다.

## 4.4 list 사용방법

list를 사용하려면 list 헤더 파일을 포함해야 한다.

```
#include <list>
```

list 형식은 아래와 같다.

```
list< 자료형 > 변수 이름
```

list를 int 형에 대해 선언한다.

```
list< int > list1;
```

선언 후에는 리스트를 사용하면 된다. 물론, 동적 할당도 가능하다.

```
list< 자료형 >* 변수 이름 = new list< 자료형 >;  
list< int >* list2 = new list< int>;
```



### 4.4.1 STL의 namespace

위에서는 list를 바로 사용했는데 이렇게 사용하려면 STL의 namespace를 선언해야 한다.

```
using namespace std;
```

위와 같이 namespace를 선언하지 않고 list를 사용하려면 STL 라이브러리 앞에 namespace를 적어 줘야 한다.

```
std::list< int > list;
```

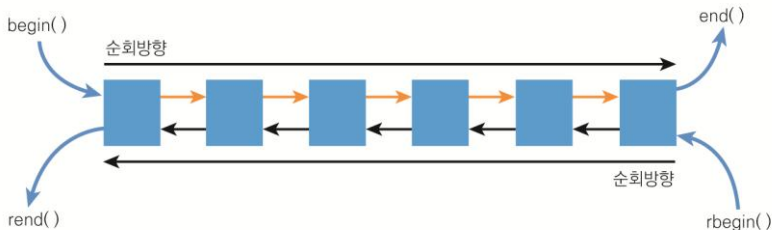
### 4.4.2 반복자(iterator)

list에 저장된 데이터에 접근하려면 반복자를 사용해야 하므로 list를 설명하기 전에 반복자에 대해서 간단하게 이야기한다. 반복자는 포인터의 일반화된 개념이라고 봐도 됩니다. STL 컨테이너에 저장된 데이터를 순회할 수 있으며 컨테이너에서 특정 위치를 가리킨다. 포인터와 비슷하게 ++과 --로 이동하고 대입과 비교도 가능하다. 그리고 각 컨테이너는 컨테이너 전용의 반복자를 구현하고 있다. 반복자의 선언 형식은 다음과 같다.

STL의 컨테이너 < 자료형 >::iterator 변수 이름

반복자 사용에 대해서 예를 들어 본다. 그림 4-6 순 방향의 앞과 끝 반복자, 역 방향의 앞과 끝 반복자

그림 4-6 반복자 사용 예



아래에 begin(), end(), rbegin(), rend()를 설명할 때, 그림 4-6을 참고한다. 설명을 위해 아래와 같이 list1을 선언한다.

```
list< int > list1;
```

## begin()

첫 번째 요소를 가리키는 반복자를 리턴한다.

예) list< int >::iterator iterFirst = list1.begin();

## end()

마지막 요소를 가리킨다. 주의할 점은 begin()과 달리 end()는 마지막 요소 바로 다음을 가리킨다. 즉 사용할 수 없는 영역을 가리키므로 end() 위치의 반복자는 사용하지 못한다.

예) list< int >::iterator iterEnd = list1.end();

for문에서 list에 저장된 모든 요소에 접근하려면 begin()과 end() 반복자를 사용하면 된다.

---

```
for( list< int >::iterator iterPos = list1.begin(); iterPos != list1.end();  
    ++iterPos )  
{  
    cout << "list1의 요소 : " << *iterPos << endl;  
}
```

---

list< int >::iterator iterPos는 list에 정의된 반복자를 가져오며, list< int >::iterator iterPos = list1.begin();은 list1의 첫 번째 요소를 가리킨다. iterPos != list1.end();는 반복자가 end()를 가리키면 for 문을 빠져 나오게 한다. ++iterPos는 반복자를 하나씩 이동시킨다.

## rbegin()

begin()와 비슷한데 다른 점은 역 방향으로 첫 번째 요소를 가리킨다는 것이다. 그리고 사용하는 반복자도 다릅니다.

예) `list::reverse_iterator IterPos = list1->rbegin();`

## rend()

`end()`와 비슷한데 다른 점은 역 방향으로 마지막 요소 다음을 가리킨다는 것이다.

예) `list::reverse_iterator IterPos = list1.rend();`

반복문에서 `rbegin()`과 `rend()`를 사용하여 `list1`의 각 데이터에 접근한다면 다음처럼 사용하면 된다.

---

```
for( list::reverse_iterator IterPos = list1.rbegin(); IterPos != list1.rend();
    ++IterPos )
{
    cout << "역 방향 list1의 요소 : " << *IterPos << endl;
}
```

---

그럼 이제 본격적으로 `list`의 주요 멤버들의 사용 법에 대해서 설명한다.

### 4.4.3 list의 주요 멤버들

표 4-1 자주 사용하는 `list` 멤버

멤버	설명
<code>begin</code>	첫 번째 위치를 가리킨다.
<code>end</code>	마지막 위치의 다음을 가리킨다.
<code>rbegin</code>	역 방향으로 첫 번째 위치를 가리킨다
<code>rend</code>	역 방향으로 마지막 위치를 가리킨다
<code>push_front</code>	첫 번째 위치에 데이터 추가
<code>pop_front</code>	첫 번째 위치의 데이터 삭제
<code>push_back</code>	마지막 위치에 데이터 추가
<code>pop_back</code>	마지막 위치의 데이터 삭제
<code>front</code>	첫 번째 데이터의 참조 리턴
<code>back</code>	마지막 데이터의 참조 리턴

clear	저장하고 있는 모든 데이터 삭제
empty	저장 데이터 유/무, 없으면 true 리턴
size	저장하고 있는 데이터의 개수 리턴
insert	지정된 위치에 삽입
erase	지정된 범위에 있는 데이터 삭제
remove	지정된 값과 일치하는 모든 데이터 삭제
remove_if	함수객체의 조건을 만족하는 모든 데이터 삭제
sort	데이터를 정렬한다.

그림 각 멤버들의 사용법에 대해서 설명한다. 다음 그림 4-7도 참조한다.

그림 4-7 list의 앞과 뒤 추가 삭제 및 접근

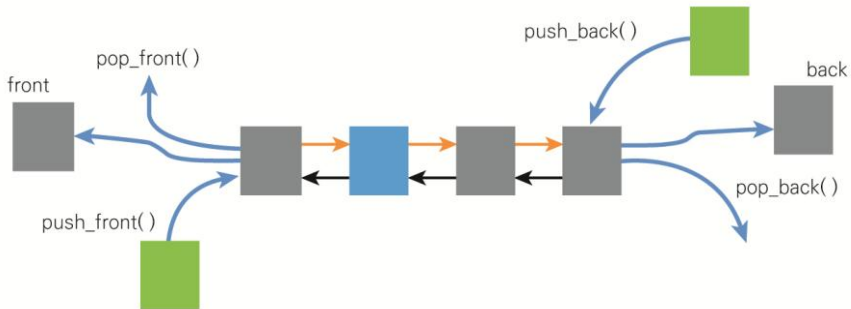


표 4-2 push\_front, pop\_front, push\_back, pop\_back, front, back, clear, empty, size의 원형 및 설명

멤버	원형	설명
push_front	void push_front(T)	첫 번째 위치에 데이터 추가
pop_front	void pop_front()	첫 번째 위치의 데이터 삭제
push_back	void push_back(T)	마지막 위치에 데이터 추가
pop_back	void pop_back()	마지막 위치에 있는 데이터 삭제
front	reference front() const_reference front() const	첫 번째 위치에 있는 데이터의 참조 리턴
back	reference back()	마지막 위치에 있는 데이터의 참조

	const_reference front() const	리턴
clear	void clear()	저장하고 있는 모든 데이터 삭제
empty	bool empty()	저장 데이터 유/무, 없으면 true 리턴
size	size_type size() const;	저장하고 있는 데이터의 개수 리턴

위에 설명한 것으로는 아직 감이 서지 않을 수도 있으니 위에 설명한 것을 사용하는 예제 코드를 참고한다.. 아래의 코드는 게임에서 사용하는 아이템의 정보를 list 컨테이너를 사용하여 아이템 정보를 앞과 뒤에 추가 및 삭제를 하고 front, back를 사용하여 저장한 아이템 요소를 출력한다.

#### [리스트 4-1]

---

```

#include <iostream>
#include <list>

using namespace std;

// 아이템 구조체
struct Item
{
    Item( int itemCd, int buyMoney )
    {
        ItemCd = itemCd;
        BuyMoney = buyMoney;
    }

    int ItemCd; // 아이템코드
    int BuyMoney; // 판매금액
};

void main()
{
    list< Item > Itemlist;

    // 앞에 데이터 추가
    Item item1( 1, 2000 );
    Itemlist.push_front( item1 );

```

---

---

```

Item item2( 2, 1000 );
Itemlist.push_front( item2 );

// 뒤에 데이터 추가
Item item3( 3, 3000 );
Itemlist.push_back( item3 );

Item item4( 4, 4500 );
Itemlist.push_back( item4 );

// 아이템 코드 번호가 2, 1, 3, 4의 순서로 출력된다.
list< Item >::iterator iterEnd = Itemlist.end();
for(list< Item >::iterator iterPos = Itemlist.begin();
    iterPos != iterEnd;
    ++iterPos )
{
    cout << "아이템 코드 : " << iterPos->ItemCd << endl;
}

// 앞에 있는 데이터를 삭제한다.
Itemlist.pop_front();

// 앞에 있는 데이터의 참조를 리턴한다.
Item front_item = Itemlist.front();
// 아이템 코드 1이 출력된다.
cout << "아이템 코드 : " << front_item.ItemCd << endl;

// 마지막에 있는 데이터를 삭제한다.
Itemlist.pop_back();

// 마지막에 있는 데이터의 참조를 리턴한다.
Item back_item = Itemlist.back();
// 아이템 코드 3이 출력된다.
cout << "아이템 코드 : " << back_item.ItemCd << endl;

// 저장된 데이터가 있는가?
if( false == Itemlist.empty() )
{

```

---

---

```

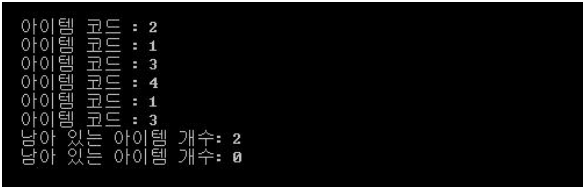
list< Item >::size_type Count = Itemlist.size();
cout << "남아 있는 아이템 개수: " << Count << endl;
}

// 모든 데이터를 지운다.
Itemlist.clear();
list< Item >::size_type Count = Itemlist.size();
cout << "남아 있는 아이템 개수: " << Count << endl;
}

```

---

[리스트 4-1]의 실행 결과는 다음과 같다.



```

아이템 개수 : 2
아이템 개수 : 1
아이템 개수 : 3
아이템 개수 : 4
아이템 개수 : 1
아이템 개수 : 3
남아 있는 아이템 개수: 2
남아 있는 아이템 개수: 0

```

그럼 계속해서 표 4-1에 소개된 list 멤버들의 설명을 계속한다.

## insert

insert는 지정된 위치에 삽입하며, 세 가지 방식이 있다. 세 가지 원형은 각각 지정된 위치에 삽입, 지정된 위치에 지정된 개수만큼 삽입, 지정된 위치에 지정 범위에 있는 것을 삽입한다. 그림 4-8을 참고한다.

---

원형 :

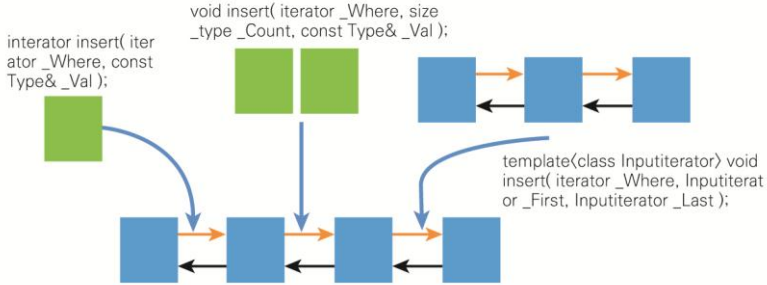
```

iterator insert( iterator _Where, const Type& _Val );
void insert( iterator _Where, size_type _Count, const Type& _Val );
template void insert( iterator _Where, InputIterator _First, InputIterator
_Last );

```

---

그림 4-8 insert의 세 가지 방법



첫 번째 insert는 지정한 위치에 데이터를 삽입한다.

---

```
list< int >::iterator iterInsertPos = list1.begin();
list1.insert( iterInsertPos, 100 ); //이 코드는 100을 첫 번째 위치에 삽입한다.
```

---

두 번째 insert는 지정한 위치에 데이터를 횟수만큼 삽입한다.

---

```
iterInsertPos = list1.begin();
++iterInsertPos;
list1.insert( iterInsertPos, 2, 200 ); //list1의 두 번째 위치에 200을 두 번
추가한다.
```

---

세 번째 insert는 지정한 위치에 복사 할 list의 시작과 끝 반복자가 가리키는 요소를 삽입한다.

---

```
list< int > list2;
list2.push_back( 1000 );
list2.push_back( 2000 );
list2.push_back( 3000 );

iterInsertPos = list1.begin();
list1.insert( ++iterInsertPos, list2.begin(), list2.end() ); // list1의 두 번째
위치에 list2의 모든 요소를 삽입한다.
```

---



다음은 위에서 설명한 insert의 세 가지 방법을 사용한 전체 코드다.

#### [리스트 4-2]

---

```
#include <iostream>
#include <list>

using namespace std;

void main()
{
    list< int > list1;

    list1.push_back(20);
    list1.push_back(30);

    cout << "삽입 테스트 1" << endl;

    // 첫 번째 위치에 삽입한다.
    list< int >::iterator iterInsertPos = list1.begin();
    list1.insert( iterInsertPos, 100 );

    // 100, 20, 30 순으로 출력된다.
    list< int >::iterator iterEnd = list1.end();
    for(list< int >::iterator iterPos = list1.begin();
        iterPos != iterEnd;
        ++iterPos )
    {
        cout << "list 1 : " << *iterPos << endl;
    }

    cout << endl << "삽입 테스트 2" << endl;

    // 두 번째 위치에 200을 2개 삽입한다.
    iterInsertPos = list1.begin();
    ++iterInsertPos;
    list1.insert( iterInsertPos, 2, 200 );
```

---

---

```

// 100, 200, 200, 20, 30 순으로출력된다.
iterEnd = list1.end();
for(list< int >::iterator iterPos = list1.begin();
    iterPos != iterEnd;
    ++iterPos )
{
    cout << "list 1 : " << *iterPos << endl;
}

cout << endl << "삽입 테스트 3" << endl;

list< int > list2;
list2.push_back( 1000 );
list2.push_back( 2000 );
list2.push_back( 3000 );

// 두 번째 위치에 list2의 모든 데이터를 삽입한다.
iterInsertPos = list1.begin();
list1.insert( ++iterInsertPos, list2.begin(), list2.end() );

// 100, 1000, 2000, 3000, 200, 200, 20, 30 순으로출력된다.
iterEnd = list1.end();
for(list< int >::iterator iterPos = list1.begin();
    iterPos != iterEnd;
    ++iterPos )
{
    cout << "list 1 : " << *iterPos << endl;
}
}

```

---

[리스트 4-2]의 실행 결과는 다음과 같다.

```
삽입 테스트 1
list 1 : 100
list 1 : 20
list 1 : 30

삽입 테스트 2
list 1 : 100
list 1 : 200
list 1 : 200
list 1 : 20
list 1 : 30

삽입 테스트 3
list 1 : 100
list 1 : 1000
list 1 : 2000
list 1 : 3000
list 1 : 200
list 1 : 200
list 1 : 20
list 1 : 30
```

## erase

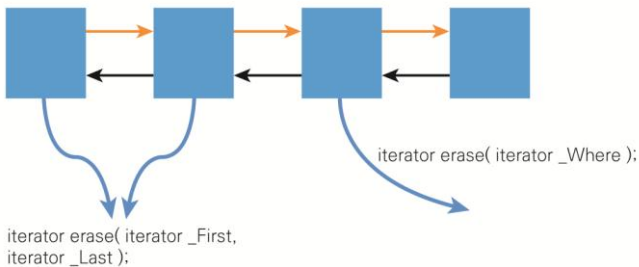
erase는 지정된 범위에 있는 데이터 삭제하며, 두 가지 방식이 있다. 하나는 지정된 위치의 데이터를 삭제하고, 다른 하나는 지정된 범위의 데이터를 삭제한다.

---

```
원형 : iterator erase( iterator _Where );
       iterator erase( iterator _First, iterator _Last );
```

---

그림 4-9 erase의 두 가지 방법



첫 번째 erase는 지정한 위치의 요소를 삭제한다. 다음은 첫 번째 요소를 삭제하는 코드다.

```
list1.erase( list1.begin() );
```

두 번째 erase는 지정한 반복자 요소만큼 삭제한다. 다음 코드는 list1의 두 번째 요소에서 마지막까지 모두 삭제한다.

---

```
list< int >::iterator iterPos = list1.begin();  
++iterPos;  
list1.erase( iterPos, list1.end() );
```

---

다음은 erase의 두 가지 사용방법을 보여주는 전체 코드다.

#### [리스트 4-3]

---

```
#include <iostream>  
#include <list>  
  
void main()  
{  
    list< int > list1;  
  
    list1.push_back(10);  
    list1.push_back(20);  
    list1.push_back(30);  
    list1.push_back(40);  
    list1.push_back(50);  
  
    cout << "erase 테스트 1" << endl;  
  
    // 첫 번째 데이터 삭제  
    list1.erase( list1.begin() );  
  
    // 20, 30, 40, 50 출력  
    list< int >::iterator iterEnd = list1.end();  
    for(list< int >::iterator iterPos = list1.begin();  
        iterPos != iterEnd;
```

---

---

```

    ++iterPos )
{
    cout << "list 1 : " << *iterPos << endl;
}

cout << endl << "erase 테스트2" << endl;

// 두 번째 데이터에서 마지막까지 삭제한다.
list< int >::iterator iterPos = list1.begin();
++iterPos;
list1.erase( iterPos, list1.end() );

// 20 출력
iterEnd = list1.end();
for(list< int >::iterator iterPos = list1.begin();
    iterPos != iterEnd;
    ++iterPos )
{
    cout << "list 1 : " << *iterPos << endl;
}
}

```

---

[리스트 4-3]의 실행 결과는 다음과 같다.



```

erase 테스트 1
list 1 : 20
list 1 : 30
list 1 : 40
list 1 : 50

erase 테스트2
list 1 : 20

```

## list 반복자의 랜덤 접근

위 에서 두 번째 위치에 접근하기 위해 ++iterPos를 사용했다. 만약 세 번째 위치로 이동하려면 한 번 더 ++iterPos를 해야 한다. list는 랜덤 접근이 안 되므로 원하는 위치까지 하나씩 이동해야 한다. 그러나 vector와 같이 랜덤 접근이 가능한 컨테이너는 다음 코드처럼 바로 접근할 수 있다.

```
iterPos = vector.begin() + 3;
```

반복문에서 list의 데이터를 삭제하면서 반복하는 경우 조심하지 않으면 버그가 발생한다. 다음 코드를 참고한다.

---

```
#include <iostream>
#include <list>

using namespace std;

void main()
{
    list< int > list1;

    list1.push_back(10);
    list1.push_back(20);
    list1.push_back(30);
    list1.push_back(40);
    list1.push_back(50);

    list< int >::iterator iterPos = list1.begin();
    while( iterPos != list1.end() )
    {
        // 3으로 나누어지는 것은 제거한다.
        if( 0 == (*iterPos % 3) )
        {
            // 삭제 되는 것의 다음 반복자를 저장하고 또 이동하지 않게 한다.
            iterPos = list1.erase( iterPos );
            continue;
        }
        cout << "list1 : " << *iterPos << endl;
        ++iterPos;
    }
}
```

---

## remove

list에서 지정한 값과 일치하는 모든 데이터 삭제. erase와 다른 점은 erase는

반복자를 통해서 삭제하지만 remove는 값을 통해서 삭제한다.

원형 : void remove( const Type& \_Val );

list1에 담겨 있는 요소 중 특정 값과 일치하는 것을 모두 삭제하고 싶을 때는 아래와 같이 한다.

list1.remove( 20 ); // 20을 삭제한다.

위에서는 값 삭제를 했지만 list가 구조체(클래스)의 포인터를 담고 있다면 삭제를 원하는 구조체의 포인터를 통해서 삭제가 가능하다. 아래는 pitem2 구조체의 포인터를 삭제한다.

---

```
// Item 포인터를 담아야 한다.
list< Item* > Itemlist;

Item* pitem1 = new Item( 10, 100 ); Itemlist.push_back( pitem1 );
Item* pitem2 = new Item( 20, 200 ); Itemlist.push_back( pitem2 );
Item* pitem3 = new Item( 30, 300 ); Itemlist.push_back( pitem3 );

// pitem2를 삭제한다.
Itemlist.remove( pitem2 );
```

---

다음은 remove의 사용법에 대한 전체 코드다.

#### [리스트 4-4]

---

```
#include <iostream>
#include <list>

using namespace std;

// 아이템 구조체
struct Item
{
    Item( int itemCd, int buyMoney )
    {
        ItemCd = itemCd;
```

---

---

```

    BuyMoney = buyMoney;
}

int ItemCd; // 아이템 코드
int BuyMoney; // 판매 금액
};

void main()
{
    list< int > list1;

    list1.push_back(10);
    list1.push_back(20);
    list1.push_back(20);
    list1.push_back(30);

    list< int >::iterator iterEnd = list1.end();
    for(list< int >::iterator iterPos = list1.begin();
        iterPos != iterEnd;
        ++iterPos )
    {
        cout << "list 1 : " << *iterPos << endl;
    }

    cout << endl << "remove 테스트 1" << endl;

    // 20을 삭제한다.
    list1.remove( 20 );

    iterEnd = list1.end();
    for(list< int >::iterator iterPos = list1.begin();
        iterPos != iterEnd;
        ++iterPos )
    {
        cout << "list 1 : " << *iterPos << endl;
    }
}

```

---



---

```

cout << endl << "remove 테스트 2 - 구조체를 삭제" << endl;

// Item 포인터를 담아야한다.
list< Item* > Itemlist;

Item* pitem1 = new Item( 10, 100 ); Itemlist.push_back( pitem1 );
Item* pitem2 = new Item( 20, 200 ); Itemlist.push_back( pitem2 );
Item* pitem3 = new Item( 30, 300 ); Itemlist.push_back( pitem3 );

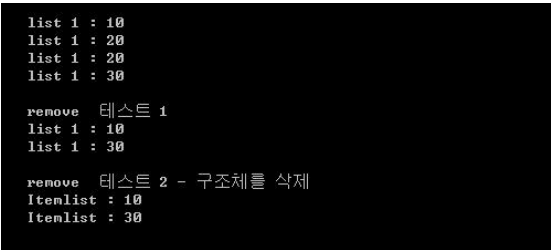
// pitem2를 삭제한다.
Itemlist.remove( pitem2 );

list< Item* >::iterator iterEnd2 = Itemlist.end();
for(list< Item* >::iterator iterPos = Itemlist.begin();
    iterPos != iterEnd2;
    ++iterPos )
{
    cout << "Itemlist : " << (*iterPos)->ItemCd << endl;
}
}

```

---

[리스트 4-4]의 실행 결과는 다음과 같다.



```

list 1 : 10
list 1 : 20
list 1 : 20
list 1 : 30

remove 테스트 1
list 1 : 10
list 1 : 30

remove 테스트 2 - 구조체를 삭제
Itemlist : 10
Itemlist : 30

```

결과 화면에서 구조체의 포인터를 담아서 삭제하는 것을 잘 살펴보기 바란다. 이미 정의된 자료형만 삭제할 수 있는 것이 아니라, 사용자 정의 타입이라도 포인터로 담으면 해당 포인터로 삭제가 가능하다.

## remove\_if

지정한 조건의 함수 객체(Predicate)를 만족하는 모든 데이터 삭제.

remove와 다른 점은 함수 객체를 사용하여 매개 변수로 전달된 인자를 조사하여 true라면 삭제하는 것이다.

참고로 함수 객체라는 것은 괄호 연산자를 멤버함수로 가지는 클래스(또는 구조체) 객체이다. 일반적으로 많이 사용되는 함수 객체는 STL에 정의되어 있다.

---

원형 : `template<class Predicate> void remove_if( Predicate _Pred );`

---

remove\_if에 사용할 함수 객체를 먼저 선언한다.

---

```
// 20 이상 30 미만이면 true
template <typename T> class Is_Over20_Under30 : public std::unary_function <T,
bool>
{
public:
    bool operator( ) ( T& val )
    {
        return ( val >= 20 && val < 30 );
    }
};
```

---

list에서 remove\_if에 함수 객체를 사용하여 list의 요소를 삭제하는 방법이다.

---

```
list< int > list1;

list1.push_back(10);
list1.push_back(20);
list1.push_back(25);
list1.push_back(30);
list1.push_back(34);

// 20 이상 30 미만은 삭제한다.
list1.remove_if( Is_Over20_Under30< int >() );
```

---

list1의 요소 중 20 이상 30 미만은 모두 삭제한다. 다음은 remove\_if 사용 예다.

#### [리스트 4-5]

---

```
#include <iostream>
#include <list>

using namespace std;

// 20 이상 30 미만이면 true
template <typename T> class Is_Over20_Under30: public std::unary_function <T,
bool>
{
public:
    bool operator( ) ( T& val )
    {
        return ( val >= 20 && val < 30 );
    }
};

void main()
{
    list< int > list1;

    list1.push_back(10);
    list1.push_back(20);
    list1.push_back(25);
    list1.push_back(30);
    list1.push_back(34);

    // 20 이상 30 미만은 삭제한다.
    list1.remove_if( Is_Over20_Under30< int >() );

    list< int >::iterator iterEnd = list1.end();
    for(list< int >::iterator iterPos = list1.begin();
        iterPos != iterEnd;
        ++iterPos )
    {
        cout << "list 1 : " << *iterPos << endl;
    }
}
```

---

---

```
}  
}
```

---

[리스트 4-5]의 결과는 다음과 같다.

```
list 1 : 10  
list 1 : 30  
list 1 : 34
```

## sort

데이터들을 정렬한다. STL에 정의된 방식으로 정렬하거나 사용자가 정의한 방식으로 정렬할 수 있다.

---

```
원형 : template<class Traits> void sort( Traits _Comp );
```

---

sort 멤버를 사용하면 list1에 있는 요소들이 올림차순으로 정렬한다.

```
list1.sort();// 올림 차순으로 정렬한다.
```

내림차순으로 정렬한다면 greater를 사용한다.

```
list1.sort( greater< int >() );
```

greater< int >는 greater< T > 라는 이미 정의되어 있는 함수 객체를 사용한 것이다. greater< int >는 int 형 x, y를 비교해서 x > y이면 true를 리턴한다.

그리고 greater외에 >=의 greater\_equal, <=의 less\_equal를 사용할 수 있다. greater 이외의 것도 사용해 보기를 바란다. 사용자 정의 함수로 정렬하려면 함수 객체를 만들어야 한다. 아래 함수 객체는 T의 멤버 중 ItemCd를 서로 비교하여 정렬을 한다.

---

```
// 함수 객체 정의  
template <typename T> struct COMPARE_ITEM  
{  
    bool operator()( const T l, const T r ) const
```

---

---

```

{
    // 정렬 시에는 올림 차순으로된다. 내림 차순으로 하고 싶으면 < 에서 > 로
    // 변경하면 된다.
    return l.ItemCd < r.ItemCd;
}
};

```

---

정의가 끝나면 다음과 같이 사용하면 됩니다.

```
Itemlist.sort( COMPARE_ITEM< Item >() );
```

Itemlist가 담고 있는 Item은 ItemCd를 기준으로 올림 차순으로 정렬한다. 아래 list의 sort 및 사용자가 정의한 함수 객체를 사용한 sort에 대한 코드다.

#### [리스트 4-6]

---

```

#include <iostream>
#include <list>

using namespace std;

// 함수 객체 정의
template <typename T> struct COMPARE_ITEM
{
    bool operator()( const T l, const T r ) const
    {
        // 정렬 시에는 올림 차순으로된다. 내림 차순으로 하고 싶으면 < 에서 > 로
        // 변경하면 된다.
        return l.ItemCd < r.ItemCd;
    }
};

// 아이템 구조체
struct Item
{
    Item( int itemCd, int buyMoney )
    {

```

---

---

```

        ItemCd = itemCd;
        BuyMoney = buyMoney;
    }

    int ItemCd; // 아이템 코드
    int BuyMoney; // 판매 금액
};

void main()
{
    list< int > list1;

    list1.push_back(20);
    list1.push_back(10);
    list1.push_back(35);
    list1.push_back(15);
    list1.push_back(12);

    cout << "sort 올림차순" << endl;
    // 올림 차순으로 정렬한다.
    list1.sort();

    list< int >::iterator iterEnd = list1.end();
    for(list< int >::iterator iterPos = list1.begin();
        iterPos != iterEnd;
        ++iterPos )
    {
        cout << "list 1 : " << *iterPos << endl;
    }

    cout << endl << "sort 내림차순" << endl;
    // 내림 차순으로 정렬한다.
    list1.sort( greater< int >() );

    iterEnd = list1.end();
    for(list< int >::iterator iterPos = list1.begin();
        iterPos != iterEnd;
        ++iterPos )

```

---

---

```

{
    cout << "list 1 : " << *iterPos << endl;
}

cout << endl << "sort - 사용자가 정의한 방식으로 정렬" << endl;

list< Item > Itemlist;

Item item1( 20, 100 ); Itemlist.push_back( item1 );
Item item2( 10, 200 ); Itemlist.push_back( item2 );
Item item3( 7, 300 ); Itemlist.push_back( item3 );

// 정렬한다.
Itemlist.sort( COMPARE_ITEM< Item >() );

list< Item >::iterator iterEnd2 = Itemlist.end();
for(list< Item >::iterator iterPos = Itemlist.begin();
    iterPos != iterEnd2;
    ++iterPos )
{
    cout << "Itemlist : " << iterPos->ItemCd << endl;
}
}

```

---

[리스트 4-6]의 실행 결과는 다음과 같다.

```

sort 오름차순
list 1 : 10
list 1 : 12
list 1 : 15
list 1 : 20
list 1 : 35

sort 내림차순
list 1 : 35
list 1 : 20
list 1 : 15
list 1 : 12
list 1 : 10

sort - 사용자가 정의한 방식으로 정렬
Itemlist : 7
Itemlist : 10
Itemlist : 20

```

보통 책에서는 list에서 제공하는 sort를 사용하는 설명이 일반적이다. 그러나 현실에서는 사용자정의 데이터를 list에 담아서 사용하므로 사용자가 정의한 함수 객체를 사용하여 정렬하는 경우가 많다.

이것으로 list에서 일반적으로 가장 자주 사용하는 멤버들에 대해서 알아 보았다. 아직 소개하지 않은 멤버들도 더 있다. 그러나 보통 현재까지 설명한 것들만 알고 있으면 list를 사용하는데 별 어려움이 없다. 소개하지 않은 멤버는 뒤에 표로 정리할 것이다.

그럼 지금까지 배운 것을 토대로 list를 사용하여 2장에서 만들었던 스택을 개선해 본다.

## 4.5 list를 사용한 스택

이전 장에 설명한 template로 만들었던 스택에 대해서 잘 기억이 나지 않으면 2장 함수 템플릿을 다시 확인한다.

이전 장에 만들었던 스택은 유연성이 부족하다. 저장 공간의 크기가 고정적이고, LIFO(후입선출. 마지막에 들어간 것이 먼저 나온다) 방식으로만 작동한다. 이것을 저장 공간의 크기가 가변적이고, FIFO(선입선출. 먼저 들어간 것이 먼저 나온다) 방식으로 저장 가능하도록 개선한다.

### [리스트 4-7]

---

```
#include <iostream>
#include <list>

using namespace std;

template<typename T>
class Stack
{
public:
    Stack() { Clear(); }
```

---



---

```
// 저장 방식을 설정한다.
void SetInOutType( bool bLIFO ) { m_bLIFO = bLIFO; }

// 초기화 한다.
void Clear()
{
    if( false == m_Datas.empty() )
        m_Datas.clear();
}

// 스택에 저장된 개수
int Count() { return static_cast( m_Datas.size() ); }

// 저장된 데이터가 없는가?
bool IsEmpty() { return m_Datas.empty(); }

// 데이터를 저장한다.
void push( T data )
{
    m_Datas.push_back( data );
}

// 스택에서 빼낸다.
bool pop( T* data )
{
    if( IsEmpty() )
    {
        return false;
    }

    if( m_bLIFO )
    {
        memcpy( data, &m_Datas.back(), sizeof(T) );
        m_Datas.pop_back();
    }
    else
    {
```

---

---

```

        memcpy( data, &m_Datas.front(), sizeof(T) );
        m_Datas.pop_front();
    }

    return true;
}

private:
    list<T> m_Datas;
    bool m_bLIFO; // true 이면 후입선출, false 이면 선입선출
};

void main()
{
    Stack< int > Int_Stack;

    // LIFO로 설정
    Int_Stack.SetInOutType( true );

    Int_Stack.push( 10 );
    Int_Stack.push( 20 );
    Int_Stack.push( 30 );

    int Value = 0;
    Int_Stack.pop( &Value );
    cout << "LIFO pop : " << Value << endl << endl;

    Int_Stack.Clear();

    // FIFO로 설정
    Int_Stack.SetInOutType( false );

    Int_Stack.push( 10 );
    Int_Stack.push( 20 );
    Int_Stack.push( 30 );

    Int_Stack.pop( &Value );

```

---

```
cout << "FIFO pop : " << Value << endl << endl;
}
```

[리스트 4-7]의 실행 결과는 다음과 같다.

```
LIFO pop : 30
FIFO pop : 10
```

list에 대해서 가장 많이 사용하는 것을 기준으로 설명했다. 보통 STL 관련 글을 보면 int나 float과 같은 기본 자료 타입을 사용하는 것에 대해서는 잘 나오지만, 사용자 정의형을 사용하는 것에 대해서는 잘 나오지 않는다. 그러므로 예제 코드 중 사용자 정의형을 사용한 부분을 특히 잘 봐야 한다. 다만, 사용자 정의형을 사용하는 경우 함수 객체라는 것을 알아야 정확하게 이해가 될 것이다. 함수 객체 부분에 대해서는 다음에 함수 객체에 대해서 설명할 때 다시 언급 할 것이다. list의 멤버는 위에서 설명한 것 이외에도 더 있다.

다른 멤버는 “<http://msdn.microsoft.com/en-us/library/00k1x78a.aspx>”를 참고한다.

표 4-3 설명하지 않은 list의 멤버들

멤버	원형	설명
assign	<pre>template&lt;class InputIterator&gt; void assign(     InputIterator _First,     InputIterator _Last );  void assign(     size_type _Count,     const Type&amp; _Val );</pre>	리스트를 저장한 값으로 채운다
get_allocator	Allocator get_allocator() const;	리스트의 할당 객체를 리턴한다
max_size	size_type max_size() const;	리스트에 저장할 수 있는 최대 크기를 리턴한다

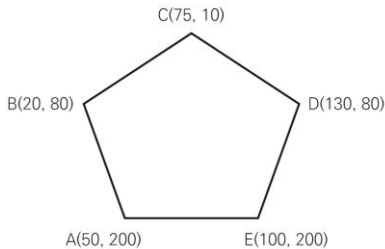
merge	<pre> void merge(     list&lt;Type, Allocator&gt;&amp; _Right ); template&lt;class Traits&gt; void merge(     list&lt;Type, Allocator&gt;&amp; _Right,     Traits _Comp ); </pre>	<p>두 리스트 요소들이 각각 정렬되어 있다면, 인수로 넘겨진 리스트의 요소와 병합한다. 정렬이 정상적으로 되어 있으면, 병합된 리스트의 요소는 정렬되어 있다.</p>
resize	<pre> void resize(     size_type _Newsize ); void resize(     size_type _Newsize,     Type _Val ); </pre>	<p>원소의 개수를 변경한다</p>
reverse	<pre> void reverse(); </pre>	<p>모든 원소의 순서를 역으로 바꾼다</p>
splice	<pre> void splice(     iterator _Where,     list&lt;Type, Allocator&gt;&amp; _Right ); void splice(     iterator _Where,     list&lt;Type, Allocator&gt;&amp; _Right,     iterator _First ); void splice(     iterator _Where,     list&lt;Type, Allocator&gt;&amp; _Right,     iterator _First,     iterator _Last ); </pre>	<p>인자로 주어진 리스트에서 지정된 범위의 요소를 대상 리스트로 이동시킨다.</p>
swap	<pre> void swap(     list&lt;Type, Allocator&gt;&amp; _Right ); friend void swap(     list&lt;Type, Allocator&gt;&amp; _Left,     list&lt;Type, Allocator&gt;&amp; _Right ) </pre>	<p>리스트간의 데이터를 교환한다</p>

unique	<pre> void unique(); template&lt;class BinaryPredicate&gt; void unique(     BinaryPredicate _Pred ); </pre>	인접한 요소 중 같은 값을 가지는 요소를 제거한다.
--------	-------------------------------------------------------------------------------------------------------------	---------------------------------

## 4.6 과제

이번 회부터는 STL 라이브러리를 설명하고 있으므로 배운 것을 활용하여 프로그램을 만들어 볼 수가 있다. 일방적으로 저의 글만 보는 것은 심심할 테니 제가 내는 문제를 풀어 보시기를 바란다.

과제그림 1 점 5개로 이루어진 도형

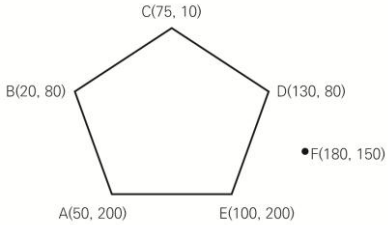


위 그림은 순서대로 A, B, C, D, E 점을 찍은 후 서로 연결하여 도형이 만들어진 것이다.

과제 1) 이것을 list를 사용하여 만든다.

꼭 그림을 그리지 않아도 된다. A, B, C, D, E의 값을 순서대로 넣고 순서대로 출력하면 된다.

과제그림 2 새로운 점 F 추가



과제 2) 점 F가 새로 추가 되었다. A, B, C, D, F, E 순으로 선이 연결 되도록 한다.

과제 3) 점 D의 값을 (200, 100)으로 변경한다.

과제 4) 점 C를 삭제한다.

아주 간단하게 list 조작을 테스트 해 볼 수 있는 간단한 과제라고 생각한다.  
꼭 프로그램을 만들어 본다.

## 5 벡터(vector)

5장에서는 vector에 대해서 이야기한다. vector는 4장에서 설명한 list와 같은 STL의 컨테이너 라이브러리이다. vector는 STL이 지원하는 자료구조 중에서도 가장 자주 사용한다. 프로그래밍을 할 때 가장 자주 사용하는 자료구조는 배열이다. vector는 배열을 대체하여 사용할 수 있다. vector는 배열과 비슷한 면이 많아서 STL 컨테이너 중에서 이해하기가 가장 쉽고 또 어디에 사용해야 하는지 알기 쉽다. 앞 장에 설명한 list와 vector는 사용방법이 비슷한 점이 많아서 list보다 훨씬 더 빠르게 이해하리라 생각한다. list에서 이미 언급한 몇몇 부분은 다시 언급하지 않으니, 앞 장의 list를 보시기 바랍니다.

### 5.1 vector의 자료구조

처음 말했듯이 vector의 자료구조는 배열과 비슷하다. Wikipedia에서 배열은 „번호(인덱스)와 번호에 대응하는 데이터로 이루어진 자료구조를 나타냅니다. 일반적으로 배열에는 같은 종류의 데이터가 순차적으로 저장된다“고 설명한다. 문자 A, B, C, D, E를 배열에 저장한다면 아래 그림과 같이 저장한다.

그림 5-1 A, B, C, D, E가 저장된 배열



배열의 크기는 고정이지만 vector는 동적으로 변하는 점이 vector와 배열 자료구조의 큰 차이점이다.

### 5.2 배열의 특징

#### 1. 배열의 크기는 고정이다.

배열은 처음에 크기를 설정하면 이후에 크기를 변경하지 못한다. 처음 설정한 크기를 넘어서 데이터를 저장할 수 없다. 그림 5-1의 배열은 A, B, C, D, E만 저장할 수 있게 5개의 크기로 만들어져 있다. 배열은 크기가 고정이므로 더 이상 새로운 것을 넣을 수 없다.

## 2. 중간에 데이터 삽입, 삭제가 용이하지 않다.

배열은 데이터를 순차적으로 저장한다. 중간에 데이터를 삽입하면 삽입한 위치 이후의 데이터는 모두 뒤로 하나씩 이동해야 한다. 또 중간에 있는 데이터를 삭제하면 삭제한 위치 이후의 데이터는 모두 앞으로 하나씩 이동해야 한다.

그림 5-2 중간에 삽입. F를 C와 D 사이에 삽입. D와 E를 뒤로 이동 시킨 후 빈 공간에 넣는다

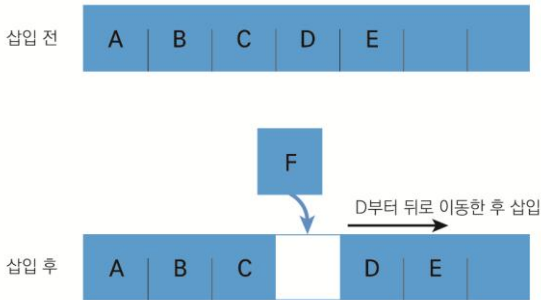
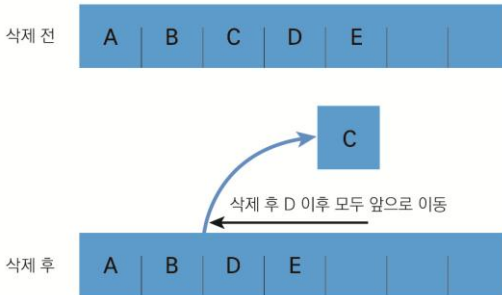


그림 5-3 중간에 삭제. C를 삭제. 삭제 후 D와 E가 앞으로 이동



## 3. 구현이 쉽다.

배열은 크기가 고정이며 중간 삭제 및 삽입에 대한 특별한 기능이 없는 아주 단순한 자료구조이다. 제일 처음 프로그래밍을 배울 때 배우는 자료구조가 배열일 정도로 구현이 쉽다.



#### 4. 랜덤 접근이 가능하다.

배열은 데이터를 순차적으로 저장하므로 랜덤 접근이 가능하다.

### 5.3 vector를 사용해야 하는 경우

#### 1. 저장할 데이터 개수가 가변적이다.

배열과 vector의 가장 큰 차이점은 „배열은 크기가 고정이고 vector는 크기가 동적으로 변한다“이다. 저장할 데이터 개수를 미리 알 수 없다면 vector를 사용 하는 편이 좋다.

#### 2. 중간에 데이터 삽입이나 삭제가 없다.

vector는 배열처럼 데이터를 순차적으로 저장한다. 중간에 데이터 삭제 및 삽입을 하면 배열과 같은 문제가 발생한다. vector는 가장 뒤에서부터 데이터를 삭제 하거나 삽입하는 경우에 적합한다.

#### 3. 저장할 데이터 개수가 적거나 많은 경우 빈번하게 검색하지 않는다.

데이터를 순차적으로 저장하므로 많은 데이터를 저장한다면 검색 속도가 빠르지 않다. 검색을 자주한다면 map이나 set, hash\_map을 사용해야 한다.

#### 4. 데이터 접근을 랜덤하게 하고 싶다.

vector는 배열 같은 특성이 있어서 랜덤 접근이 가능하다. 특정 데이터가 저장된 위치를 안다면 랜덤 접근을 사용하는 쪽이 성능이 좋고, 사용하기도 간편합니다. 예를 들면 온라인 게임 제작 시 아이템 번호를 순차적으로 부여한다고 가정한다. 아이템 데이터를 vector에 저장하면 아이템 개수가 늘어나더라도 코드를 수정하지 않아도 되며, 아이템 코드 7번은 언제나 7번째 위치에 있으므로 랜덤 접근으로 빠르고 쉽게 접근할 수 있다. 위에 열거한 배열의 특징과 vector의 특징을 잘 숙지하여 기존에 배열을 사용한 부분에 vector를 사용하면 배열의 단점을 없앤 유지보수성이 좋은 코드를 만들게 됩니다.

### 5.4 vector vs. list

vector 사용법을 보면 list와 비슷한 부분도 있고 다른 부분도 있음을 알게 되리라 생각한다. vector와 list의 차이점을 잘 이해한 후 올바르게 사용해야 됩니다 . vector와 list의 차이를 정리하면 아래 표와 같다.

표 5-1 vector와 list의 차이

	vector	List
크기 변경 가능	O	O
중간 삽입, 삭제 용이	X	O
순차 접근 가능	O	O
랜덤 접근 가능	O	X

표 5-1을 보시면 아시겠지만 vector와 list의 차이점은 크게 2가지다.

- 중간 삽입, 삭제
- 랜덤 접근

중간 삽입, 삭제가 없고 랜덤 접근을 자주 해야 된다면 vector가 좋고, 중간 삽입, 삭제가 자주 있으며 랜덤 접근이 필요 없으면 list가 좋다.

**NOTE** 중간 삽입 삭제가 있다면 무조건 list를 사용해야 할까요? list와 vector의 차이점을 보면 중간 삽입, 삭제가 자주 일어나는 자료구조는 list 사용이 정답이다. 그렇다고 list 사용이 항상 정답은 아닙니다. 만약 저장하는 데이터의 개수가 적고 랜덤 접근을 하고 싶은 경우에는 vector를 사용해도 좋다.

필자가 하는 일을 예를 들면 대부분의 온라인 캐주얼 게임은 방을 만들어서 방에 들어온 유저끼리 게임을 한다. 방은 유저가 들어 오기도 하고 나가기도 한다. 중간 삽입, 삭제가 자주 일어나지만 방의 유저 수는 대부분 적다. 이런 경우 중간 삽입, 삭제로 데이터를 이동해도 전체적인 성능 측면에서는 문제가 되지 않다. 방에 있는 유저를 저장한 위치를 알고 있다면 유저 데이터에 접근할 때 list는 반복문으로 해당 위치까지 순차적으로 접근해야 하지만 vector는 바로 랜덤 접근이 가능하고 성능면에서도 더 좋다. 또한, 방에 있는 모든 유저 데이터에 접근할 때 list는 반복자(Iterator)로 접근하지만, vector는 일반 배열처럼 접근하므로 훨씬 간단하게 유저 데이터에 접근할 수 있다.

저장하는 데이터 개수가 적은 경우는 대부분 list 보다는 vector를 사용하는 편이 더 좋다. vector와 list의 차이점만으로 둘 중 어느 것을 사용할지 선택하기 보다는 전체적인 장, 단점을 파악하고나서 선택하는 것이 좋다.

## 5.5 vector 사용방법

vector를 사용하려면 vector 헤더 파일을 포함해야 한다.

```
#include <vector>
```

vector 형식은 아래와 같다.

```
vector< 자료형> 변수 이름
```

vector를 int 형에 대해 선언했다.

```
vector< int > vector1;
```

선언 후 vector를 사용한다. vector도 list처럼 동적 할당이 가능하다.

```
vector< 자료형 >* 변수 이름 = new vector< 자료형 >;  
vector< int >* vector1 = new vector< int>;
```

## 5.6 vector의 주요 멤버들

vector 멤버 중 일반적으로 자주 사용하는 멤버들은 다음과 같다.

표 5-2 자주 사용하는 vector 멤버

멤버	설명
assign	특정 원소로 채운다
at	특정 위치의 원소의 참조를 리턴
back	마지막 원소의 참조를 리턴
begin	첫 번째 원소의 랜덤 접근 반복자를 리턴
clear	모든 원소를 삭제
empty	아무것도 없으면 true 리턴
end	마지막 원소 다음의(사용하지 않은 영역) 반복자를 리턴
erase	특정 위치의 원소나 지정 범위의 원소를 삭제
front	첫 번째 원소의 참조를 리턴

insert	특정 위치에 원소 삽입
pop_back	마지막 원소를 삭제
push_back	마지막에 원소를 추가
rbegin	역방향으로 첫 번째 원소의 반복자를 리턴
rend	역방향으로 마지막 원소 다음의 반복자를 리턴
reserve	지정된 크기의 저장 공간을 확보
size	원소의 개수를 리턴
swap	vector 의 두 원소를 서로 교환한다

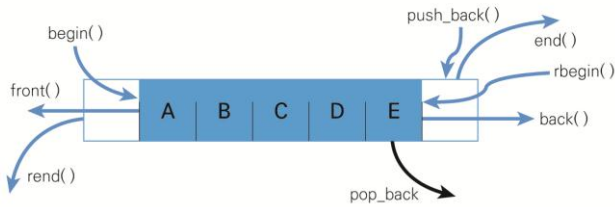
## 5.6.1 기본 사용 멤버

vector의 가장 기본적인(추가, 삭제, 접근 등) 사용법을 설명 하겠다. 표 5-3과 그림 5-4를 참고 해주세요

표 5-3 추가, 삭제, 접근 등에 사용하는 멤버들

멤버	원형	설명
at	reference at( size_type _Pos ); const_reference at( size_type _Pos ) const;	특정 위치의 원소의 참조를 리턴경계 체크 기능이 있음
back	reference back( ); const_reference back( ) const;	마지막 원소의 참조를 리턴
begin	const_iterator begin() const; iterator begin();	첫 번째 원소의 랜덤 접근 반복자를 리턴
clear	void clear();	모든 원소를 삭제
empty	bool empty() const;	아무것도 없으면 true 리턴
end	iterator end( ); const_iterator end( ) const;	마지막 원소 다음의(사용하지 않은 영역) 반복자를 리턴
front	reference front( ); const_reference front( ) const;	첫 번째 원소의 참조를 리턴
pop_back	void pop_back();	마지막 원소를 삭제
push_back	void push_back( const Type& _Val );	마지막에 원소를 추가
rbegin	reverse_iterator rbegin( ); const_reverse_iterator rbegin( ) const;	역방향으로 첫 번째 원소의 반복자를 리턴
rend	const_reverse_iterator rend( ) const; reverse_iterator rend( );	역방향으로 마지막 원소 다음의 반복자를 리턴
size	size_type size() const;	원소의 개수를 리턴

그림 5-4 vector의 추가, 삭제, 접근 멤버를 나타내고 있다



## 추가

기본적으로 원소의 마지막 위치에 추가하며 `push_back`을 사용한다. 처음이나 중간 위치에 추가할 때는 다음에 소개할 `insert`를 사용한다.

```
vector< int > vector1;  
vector1.push_back( 1 );
```

## 삭제

기본적으로 마지막 위치의 원소를 삭제하며 `pop_back`을 사용한다. 처음이나 중간에 있는 원소를 삭제할 때는 다음에 소개할 `erase`를 사용한다.

```
vector1.pop_back();
```

## 접근

첫 번째 위치의 반복자를 리턴할 때는 `begin()`을 사용한다. 첫 번째 원소의 참조를 리턴할 때는 `front()`를 사용한다.

---

```
vector< int >::iterator IterBegin = vector1.begin();  
cout << *IterBegin << endl;  
int& FirstValue = vector1.front();  
const int& reffFirstValue = vector1.front();
```

---

마지막 다음의 영역(사용하지 않는 영역)을 가리키는 반복자를 리턴할 때는 end()를 사용한다. 마지막 원소의 참조를 리턴할 때는 back()을 사용한다.

---

```
vector< int >::iterator IterEnd = vector1.end();
for(vector< int >::iterator IterPos = vector1.begin(); IterPos != vector1.end();
++IterPos )
{
    .....
}

int& LastValue = vector1.back();
const int& refLastValue = vector1.back();
```

---

rbegin()과 rend()는 방향이 역방향이라는 점만 다를 뿐, 나머지는 begin()과 end()와 같다. 특정 위치에 있는 원소를 접근할 때는 at()을 사용하면 됩니다.

```
int& Value1 = vector1.at(0); // 첫 번째 위치
const int Value2 = vector1.at(1); // 두 번째 위치
```

배열 식 접근도 가능하다. vector를 사용할 때 보통 이 방식으로 자주 사용한다.

```
int Value = vector1[0]; // 첫 번째 위치
```

## 모두 삭제

저장한 모든 데이터를 삭제할 때는 clear()를 사용한다.

```
vector1.clear();
```

## 데이터 저장 여부

vector에 저장한 데이터가 있는지 없는지는 empty()로 조사한다. empty()는 데이터가 있으면 false, 없다면 true를 리턴한다.

```
bool bEmpty = vector1.empty();
```

## vector에 저장된 원소 개수 알기

size()를 사용하여 vector에 저장 되어 있는 원소 개수를 조사한다.

```
vector< int >::size_type Count = vector1.size();
```

위에 설명한 멤버들을 사용하는 아래의 예제 코드를 보면서 vector의 기본 사용 방법을 정확하게 숙지하길 바란다.

### [리스트 5-1] 온라인 게임의 게임 방의 유저 관리

---

```
#include <vector>
#include <iostream>

using namespace std;

// 방의 유저 정보
struct RoomUser
{
    int CharCd; // 캐릭터 코드
    int Level;  // 레벨
};

void main()
{
    // 유저1
    RoomUser RoomUser1;
    RoomUser1.CharCd = 1;
    RoomUser1.Level = 10;

    // 유저2
    RoomUser RoomUser2;
    RoomUser2.CharCd = 2;
    RoomUser2.Level = 5;

    // 유저3
    RoomUser RoomUser3;
```

---

---

```
RoomUser3.CharCd = 3;
RoomUser3.Level = 12;

// 방의 유저들을 저장할 vector
vector< RoomUser > RoomUsers;

// 추가
RoomUsers.push_back( RoomUser1 );
RoomUsers.push_back( RoomUser2 );
RoomUsers.push_back( RoomUser3 );

// 방에 있는 유저 수
int UserCount = RoomUsers.size();

// 방에 있는 유저 정보 출력
// 반복자로 접근 - 순방향
for( vector< RoomUser >::iterator IterPos = RoomUsers.begin();
    IterPos != RoomUsers.end();
    ++IterPos )
{
    cout << "유저코드 : " << IterPos->CharCd << endl;
    cout << "유저레벨 : " << IterPos->Level << endl;
}
cout << endl;

// 반복자로 접근- 역방향
for( vector< RoomUser >::reverse_iterator IterPos = RoomUsers.rbegin();
    IterPos != RoomUsers.rend();
    ++IterPos )
{
    cout << "유저코드: " << IterPos->CharCd << endl;
    cout << "유저레벨: " << IterPos->Level << endl;
}
cout << endl;

// 배열 방식으로 접근
for( int i = 0; i < UserCount; ++i )
{
```

---



---

```
        cout << "유저 코드 : " << RoomUsers[i].CharCd << endl;
        cout << "유저 레벨 : " << RoomUsers[i].Level << endl;
    }
    cout << endl;

    // 첫 번째 유저 데이터 접근
    RoomUser& FirstRoomUser = RoomUsers.front();
    cout << "첫 번째 유저의 레벨 : " << FirstRoomUser.Level << endl << endl;

    RoomUser& LastRoomUser = RoomUsers.back();
    cout << "마지막 번째 유저의 레벨: " << LastRoomUser.Level << endl << endl;

    // at을 사용하여 두 번째 유저의 레벨을 출력
    RoomUser& RoomUserAt = RoomUsers.at(1);
    cout << "두 번째 유저의 레벨: " << RoomUserAt.Level << endl << endl;

    // 삭제
    RoomUsers.pop_back();

    UserCount = RoomUsers.size();
    cout << "현재 방에 있는 유저 수: " << UserCount << endl << endl;

    // 아직 방에 유저가 있다면 모두 삭제한다.
    if( false == RoomUsers.empty() )
    {
        RoomUsers.clear();
    }

    UserCount = RoomUsers.size();
    cout << "현재 방에 있는 유저 수: " << UserCount << endl;
}
```

---

[리스트 5-1]의 실행 결과는 다음과 같다.

```
유저 코드 : 1
유저 레벨 : 10
유저 코드 : 2
유저 레벨 : 5
유저 코드 : 3
유저 레벨 : 12

유저 코드 : 3
유저 레벨 : 12
유저 코드 : 2
유저 레벨 : 5
유저 코드 : 1
유저 레벨 : 10

유저 코드 : 1
유저 레벨 : 10
유저 코드 : 2
유저 레벨 : 5
유저 코드 : 3
유저 레벨 : 12

첫 번째 유저의 레벨 : 10
마지막 번째 유저의 레벨 : 12
두 번째 유저의 레벨 : 5
현재 방에 있는 유저 수 : 2
현재 방에 있는 유저 수 : 0
```

혹시 랜덤 접근이 무엇인지 잘 이해하지 못한 분은 [리스트 5-1]의 예제 코드에서 배열 방식으로 접근하는 부분을 잘 보세요. 배열처럼 접근 하는 것을 랜덤 접근이 가능하다고 말한다. 랜덤 접근이 안 되는 list에서는 오직 반복자로 순차 접근만 가능하다.

## insert

insert는 지정된 위치에 삽입하며, 세 가지 방식이 있다. list의 insert와 사용 방법이 같다. 세 가지 원형은 각각 지정한 위치에 삽입, 지정한 위치에 지정된 개수만큼 삽입, 지정한 위치에 지정 범위에 있는 것을 삽입한다. vector의 insert를 사용할 때에는 삽입한 위치 이후의 원소들이 모두 뒤로 이동함을 꼭 숙지하여야 합니다.

---

```
원형 : iterator insert( iterator _Where, const Type& _Val );
      void insert( iterator _Where, size_type _Count, const Type& _Val );
```

---

---

```
template<class InputIterator> void insert( iterator _Where,
InputIterator _First,                      InputIterator _Last );
```

---

그림 5-5 vector의 insert

**a**      iterator insert( iterator \_Where, const Type& \_Val );

삽입 전      A | B | C | D | E

삽입 후      A | A | B | C | D | E

void insert( iterator \_Where, size\_type \_Count, const Type& \_Val );

**a** | **b**

삽입 전      A | B | C | D | E

삽입 후      A | B | a | b | C | D | E

template<class InputIterator> void insert( iterator \_Where,
InputIterator \_First, InputIterator \_Last );

**a** | **b** | **c**

삽입 전      A | B | C | D | E

삽입 후      A | B | a | b | c | C | D | E

첫 번째 insert는 지정한 위치에 데이터를 삽입한다.

---

```
vector< int >::iterator iterInsertPos = vector1.begin();
vector1.insert( iterInsertPos, 100 ); //이 코드는 100을 첫 번째 위치에 삽입한다.
```

---

두 번째 insert는 지정한 위치에 데이터를 횡수만큼 삽입한다.

---

```
iterInsertPos = vector1.begin();
++iterInsertPos;
vector1.insert( iterInsertPos, 2, 200 );//두 번째 위치에 200을 두 번 추가한다.
```

---

세 번째 insert는 지정한 위치에 복사 할 vector의 (복사하기를 원하는 영역의)시작과 끝 반복자가 가리키는 영역의 모든 요소를 삽입한다.

---

```
vector< int > vector2;
list2.push_back( 500 );
list2.push_back( 600 );
list2.push_back( 700 );

iterInsertPos = vector1.begin();
vector1.insert( ++iterInsertPos, vector2.begin(), vector2.end() );
```

---

vector1의 두 번째 위치에 vector2의 모든 요소를 삽입한다. 위에서 설명한 insert의 세 가지 방법을 사용한 전체 코드이다. 이 예제는 4장, list 의 insert 에서 소개했던 코드와 같다. 오직 list 대신 vector를 사용했다는 것만 다르다.

#### [리스트 5-2] insert 사용

---

```
void main()
{
    vector< int > vector1;

    vector1.push_back(20);
    vector1.push_back(30);

    cout << "삽입 테스트 1" << endl;

    // 첫 번째 위치에 삽입한다.
    vector< int >::iterator iterInsertPos = vector1.begin();
    vector1.insert( iterInsertPos, 100 );
```

---

---

```

// 100, 20, 30 순으로 출력한다.
vector< int >::iterator iterEnd = vector1.end();
for(vector< int >::iterator iterPos = vector1.begin();
    iterPos != iterEnd;
    ++iterPos )
{
    cout << "vector1 : " << *iterPos << endl;
}

cout << endl << "삽입 테스트 2" << endl;

// 두 번째 위치에 200을 2개 삽입한다.
iterInsertPos = vector1.begin();
++iterInsertPos;
vector1.insert( iterInsertPos, 2, 200 );

// 100, 200, 200, 20, 30 순으로 출력한다.
iterEnd = vector1.end();
for(vector< int >::iterator iterPos = vector1.begin();
    iterPos != iterEnd;
    ++iterPos )
{
    cout << "vector1 : " << *iterPos << endl;
}

cout << endl << "삽입 테스트 3" << endl;

vector< int > vector2;
vector2.push_back( 1000 );
vector2.push_back( 2000 );
vector2.push_back( 3000 );

// 두 번째 위치에 vector2의 모든 데이터를 삽입한다.
iterInsertPos = vector1.begin();
vector1.insert( ++iterInsertPos, vector2.begin(), vector2.end() );

// 100, 1000, 2000, 3000, 200, 200, 20, 30 순으로 출력한다.

```

---

---

```

iterEnd = vector1.end();
for(vector< int >::iterator iterPos = vector1.begin();
    iterPos != iterEnd;
    ++iterPos )
{
    cout << "vector1 : " << *iterPos << endl;
}
}

```

---

[리스트 5-2]의 실행 결과는 다음과 같다.

```

삽입 테스트 1
vector1 : 100
vector1 : 20
vector1 : 30

삽입 테스트 2
vector1 : 100
vector1 : 200
vector1 : 200
vector1 : 20
vector1 : 30

삽입 테스트 3
vector1 : 100
vector1 : 1000
vector1 : 2000
vector1 : 3000
vector1 : 200
vector1 : 200
vector1 : 20
vector1 : 30

```

## erase

반복자로 특정 위치의 요소를 삭제할 때는 erase를 사용한다. 사용 방식은 두 가지가 있다. 하나는 지정한 위치의 요소를 삭제하고, 다른 하나는 지정한 범위의 요소를 삭제한다. 마지막 위치 이외의 곳에서 erase를 할 때는 삭제한 위치 이후의 모든 원소들이 앞으로 이동한다는 것을 꼭 숙지하셔야 됩니다.

---

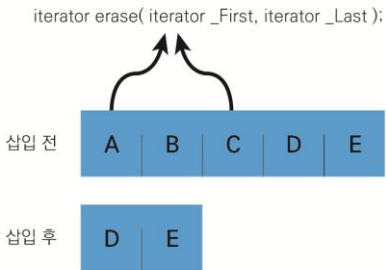
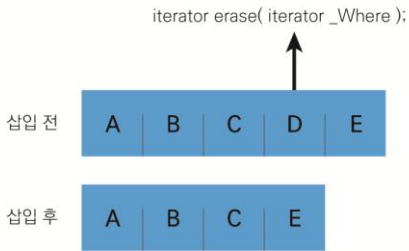
```

원형 : iterator erase( iterator _Where );
       iterator erase( iterator _First, iterator _Last );

```

---

그림 5-6 vector의 erase



첫 번째 erase는 지정한 위치의 요소를 삭제한다. 다음은 첫 번째 요소를 삭제하는 코드다.

```
vector1.erase( vector1.begin() );
```

두 번째 erase는 지정한 반복자 요소만큼 삭제한다. 다음 코드는 vector1의 첫 번째 요소에서 마지막까지 모두 삭제한다.

```
vector1.erase( vector1.begin(), vector1.end() );
```

다음은 erase 사용을 보여주는 예제다.

### [리스트 5-3] erase 사용

---

```
void main()
{
    vector< int > vector1;

    vector1.push_back(10);
    vector1.push_back(20);
    vector1.push_back(30);
    vector1.push_back(40);
    vector1.push_back(50);

    int Count = vector1.size();
    for( int i = 0; i < Count; ++i )
    {
        cout << "vector 1 : " << vector1[i] << endl;
    }
    cout << endl;

    cout << "erase 테스트 1" << endl;

    // 첫 번째 데이터 삭제
    vector1.erase( vector1.begin() );

    // 20, 30, 40, 50 출력
    Count = vector1.size();
    for( int i = 0; i < Count; ++i )
    {
        cout << "vector 1 : " << vector1[i] << endl;
    }

    cout << endl << "erase 테스트" << endl;

    // 첫 번째 데이터에서 마지막까지 삭제한다.
    vector< int >::iterator iterPos = vector1.begin();
```

---



---

```

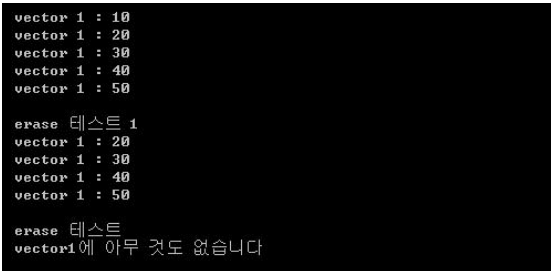
vector1.erase( iterPos, vector1.end() );

if( vector1.empty() )
{
    cout << "vector1에 아무 것도 없습니다" << endl;
}
}

```

---

[리스트 5-3]의 실행 결과는 다음과 같다.



```

vector 1 : 10
vector 1 : 20
vector 1 : 30
vector 1 : 40
vector 1 : 50

erase 테스트 1
vector 1 : 20
vector 1 : 30
vector 1 : 40
vector 1 : 50

erase 테스트
vector1에 아무 것도 없습니다

```

## assign

vector를 어떤 특정 데이터로 채울 때는 assign을 사용하면 됩니다. 사용 방식은 두 가지가 있다. 첫 번째는 특정 값으로 채우는 방법이고, 두 번째는 다른 vector의 반복자로 지정한 영역에 있는 원소로 채우는 방법이다. 만약 assign을 사용한 vector에 이미 데이터가 있다면 기존의 것은 모두 지우고 채운다.

---

```

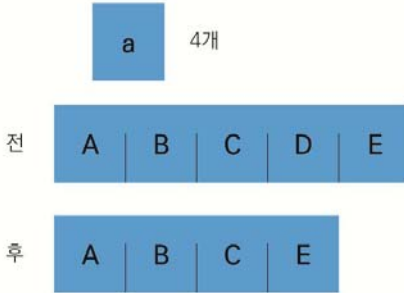
원형 : void assign( size_type _Count, const Type& _Val );
        template<class InputIterator> void assign( InputIterator _First,
InputIterator _Last );

```

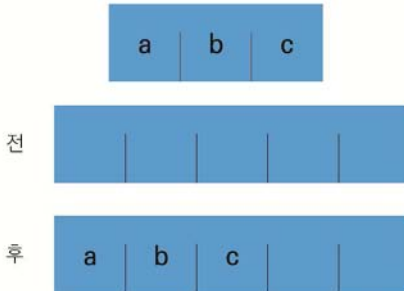
---

## 그림 5-7 assign

```
void assign( size_type _Count, const Type& _Val );
```



```
template<class Inputiterator> void assign( Inputiterat  
or _First, Inputiterator _Last );
```



첫 번째 assign은 지정 데이터를 지정 개수만큼 채워줍니다. 숫자 4를 7개 채운다.

```
vector1.assign( 7, 4 );
```

두 번째 assign은 다른 vector의 반복자로 지정한 영역으로 채워줍니다.

```
vector1.erase( vector1.begin(), vector1.end() );
```

다음은 assign을 사용법을 보여주는 예제다.

#### [리스트 5-4] assign

---

```
void main()
{
    vector< int > vector1;

    // 4를 7개 채운다.
    vector1.assign( 7, 4 );

    int Count = vector1.size();
    for( int i = 0; i < Count; ++i )
    {
        cout << "vector 1 : " << vector1[i] << endl;
    }
    cout << endl;

    vector< int > vector2;
    vector2.push_back(10);
    vector2.push_back(20);
    vector2.push_back(30);

    // vector2의 요소로 채운다
    vector1.assign( vector2.begin(), vector2.end() );
    Count = vector1.size();
    for( int i = 0; i < Count; ++i )
    {
        cout << "vector 1 : " << vector1[i] << endl;
    }
    cout << endl;
}
```

---

[리스트 5-4]의 실행 결과는 다음과 같다.

```
vector 1 : 4
vector 1 : 4
vector 1 : 4
vector 1 : 4
vector 1 : 4
vector 1 : 4
vector 1 : 4

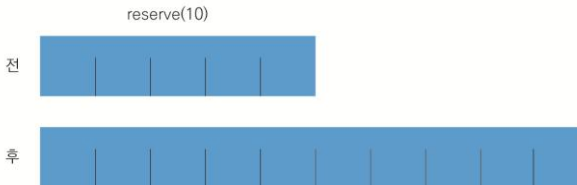
vector 1 : 10
vector 1 : 20
vector 1 : 30
```

## reserve

vector는 사용할 메모리 영역을 처음 선언할 때 정해진 값만큼 할당한 후 이 크기를 넘어서게 사용하면 현재 할당한 크기의 2배의 크기로 재할당한다. vector에 어느 정도의 데이터를 저장할지 예측할 수 있고, vector 사용 도중에 재할당이 일어나는 것을 피하려면 사용할 만큼의 크기를 미리 지정해야 한다. 참고로 reserve로 지정할 수 있는 크기는 vector에서 할당하는 최소의 크기보다는 커야 한다.

원형 : void reserve( size\_type \_Count );

그림 5-8 reserve



10개의 원소를 채울 수 있는 공간 확보

```
vector1.reserve( 10 );
```

## swap

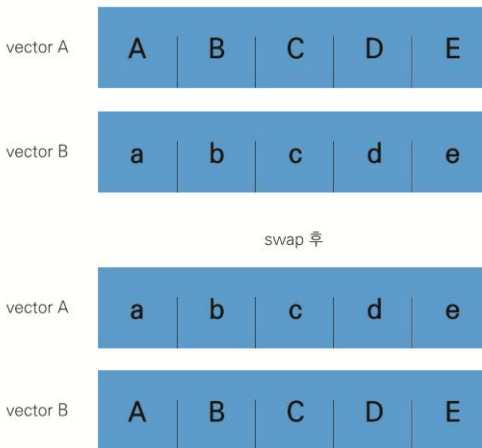
vector1과 vector2가 있을 때 두 개의 vector간에 서로 데이터를 맞바꾸기를 할 때 사용한다.

---

```
원형 : void swap( vector<Type, Allocator>& _Right );  
        friend void swap( vector<Type, Allocator >& _Left, vector<Type,  
Allocator >& _Right );
```

---

그림 5-9 swap



vector1과 vector2를 swap

```
vector1.swap( vector2 );
```

swap을 사용하는 예제다.

### [리스트 5-5] Swap

---

```
void main()  
{
```

---

---

```
vector< int > vector1;
vector1.push_back(1);
vector1.push_back(2);
vector1.push_back(3);

vector< int > vector2;
vector2.push_back(10);
vector2.push_back(20);
vector2.push_back(30);
vector2.push_back(40);
vector2.push_back(50);

int Count = vector1.size();
for( int i = 0; i < Count; ++i )
{
    cout << "vector 1 : " << vector1[i] << endl;
}
cout << endl;

Count = vector2.size();
for( int i = 0; i < Count; ++i )
{
    cout << "vector 2 : " << vector2[i] << endl;
}
cout << endl;
cout << endl;

cout << "vector1과 vector2를 swap" << endl;

vector1.swap(vector2);

Count = vector1.size();
for( int i = 0; i < Count; ++i )
{
    cout << "vector 1 : " << vector1[i] << endl;
}
cout << endl;
```

---

---

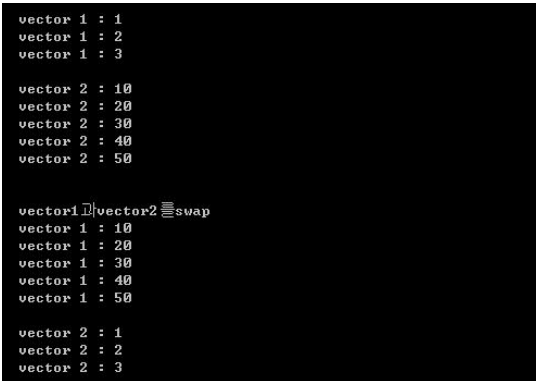
```

Count = vector2.size();
for( int i = 0; i < Count; ++i )
{
    cout << "vector 2 : " << vector2[i] << endl;
}
}

```

---

[리스트 5-5]의 실행 결과는 다음과 같다.



```

vector 1 : 1
vector 1 : 2
vector 1 : 3

vector 2 : 10
vector 2 : 20
vector 2 : 30
vector 2 : 40
vector 2 : 50

vector1과vector2를swap
vector 1 : 10
vector 1 : 20
vector 1 : 30
vector 1 : 40
vector 1 : 50

vector 2 : 1
vector 2 : 2
vector 2 : 3

```

vector 중에서 가장 많이 사용하는 멤버를 중심으로 설명 하였다. vector의 모든 멤버를 설명하지는 않았으니 소개하지 않은 나머지 멤버까지 알고 싶다면 마이크로소프트의 MSDN에 나와 있는 것을 참고한다.

<http://msdn.microsoft.com/en-us/library/sxcsf7y7.aspx>

이번에 설명한 vector에서 소개한 front(), push\_back(), pop\_back(), erase() 등이 list의 멤버들과 사용방법이나 결과가 같음을 알 수 있다. 이것이 STL를 사용하여 얻는 장점 중의 하나이다.

STL에서 제공하는 컨테이너들은 서로 특성은 다르지만 사용방법과 결과가 같기 때문에 하나만 잘 알면 다른 것들도 쉽게 배울 수 있다. 만약 STL의 컨테이너를 사용하지 않고 독자적으로 구현하여 사용한다면 각각 사용방법이 달라서 사용방

법을 배울 때마다 STL보다 더 많은 시간이 필요할 것이며, 함수 이름을 보고 어떤 동작을 할지 각각의 라이브러리마다 숙지해야 하므로 유지보수에 좋지 않다.

vector는 배열과 비슷하고 사용하기 편리하여 많은 곳에서 사용한다. 그러나 vector의 특성을 제대로 이해하지 못하고 잘못된 곳에 사용하면 심각한 성능 저하가 일어날 수 있다(많은 데이터를 저장하고 있으며 빈번하게 중간에서 삽입, 삭제를 할 때). 그러니 꼭 적합한 장소에 사용해야 한다.

## 5.7 과제

1. “4.5 list를 사용한 스택”에서 list를 사용하여 LIFO 방식으로 스택을 만든 예제가 있는데 이것을 vector를 사용하여 만들어 보세요
2. „카트 라이더”와 같이 방을 만들어서 게임을 하는 온라인 게임에서 방에 있는 유저를 관리하는 부분을 vector를 사용하여 만들어 보세요. 기본적인 클래스 선언은 제시할 테니 구현만 하면 됩니다.

---

```
// 유저 정보
struct UserInfo
{
    char acUserName[21]; // 이름
    intLevel;           // 레벨
    int Exp;            // 경험치
};

// 게임 방의 유저를 관리하는 클래스
// 방에는 최대 6명까지 들어 갈 수 있다.
// 방에 들어 오는 순서 중 가장 먼저 들어 온 사람이 방장이 된다.
class GameRoomUser
{
public:
    GameRoomUser();
    ~GameRoomUser();

    // 방에 유저 추가
    bool AddUser( UserInfo& tUserInfo );
```

---



---

```
// 방에서 유저 삭제.
// 만약 방장이 나가면 acMasterUserName에 새로운 방장의 이름을 설정 해야 된다.
bool DelUser( char* pcUserName );

// 방에 유저가 있는지 조사. 없으면 true 리턴
bool IsEmpty();

// 방에 유저가 꽉 찼는지 조사. 꽉 찼다면 true 리턴
bool IsFull();

// 특정 유저의 정보
UserInfo& GetUserOfName( char* pcName );

// 방장의 유저 정보
UserInfo& GetMasterUser();

// 가장 마지막에 방에 들어 온 유저의 정보
UserInfo& GetUserOfLastOrder();

// 특정 순서에 들어 온 유저를 쫓아낸다.
bool BanUser( int OrderNum );

// 모든 유저를 삭제한다.
void Clear();

private:
    vector< UserInfo > Users;
    char acMasterUserName[21]; // 방장의 이름
};
```

---

## 6 덱(deque)

이 장에서는 STL 컨테이너 라이브러리 중 하나인 deque을 설명한다. 앞에서 list, vector에 대한 글을 본 독자들은 알겠지만 STL 컨테이너 라이브러리는 사용하는 방법이 서로 비슷하므로 하나만 잘 알면 다른 컨테이너도 쉽게 배울 수 있다. 4장 list나 5장 vector에 대한 글을 보지 않았다면 꼭 먼저 보고나서 이 장을 보도록 한다. 이미 list나 vector를 알고 있는 독자의 경우 deque의 자료구조 및 특징을 잘 파악하기를 바란다.

### 6.1 deque의 자료구조

deque의 자료구조는 이름과 같이 Deque<sup>Double Ended Queue</sup> 자료구조다. Deque 자료구조는 큐<sup>Queue</sup> 자료구조와 비슷하므로, 먼저 큐 자료구조를 설명한다. Queue는 선형 리스트로 선입선출(FIFO) 방식을 사용한다. 다음 그림처럼 시작과 끝을 가리키는 포인터로 삽입과 삭제를 한다.

그림 6-1 Queue 자료구조



그림 6-1에서 F(Front)는 가장 앞에 있는 것을 가리키며 삭제 작업을 할 때 사용한다. 위 그림에서 삭제를 하면 „a”는 없어지고 F는 „b”를 가리킨다. R(Rear)은 가장 마지막에 있는 것을 가리키며 삽입 작업을 할 때 사용한다. 위 그림에서 „g”를 추가하면 „f” 다음에 위치하며 R은 새로 추가된 „g”를 가리킨다.

Queue는 앞으로는 삭제, 뒤로는 삽입을 할 때 사용한다.

OS의 작업 스케줄링처럼 입력 순서대로 처리를 할 때 큐를 사용하면 좋다. 덱<sup>Deque</sup>이 Queue와 다른 점은 삽입과 삭제를 한쪽이 아닌 앞, 뒤 양쪽에서 할 수 있다는 것이며, 다른 것은 Deque과 같다.

그림 6-2 덱 자료구조



그림 6-1과 다르게 그림 6-2를 보면 앞과 뒤에서 삽입과 삭제를 할 수 있다. Deque은 Stack과 Queue의 장점을 모은 것으로 FIFO 방식과 LIFO 방식 둘 다 사용할 수 있다.

## 6.2 Deque의 특징

Deque의 장점을 정리하면 다음과 같다.

1. 크기가 가변적이다.  
리스트와 같이 데이터를 담을 수 있는 크기가 가변적이다.
2. 앞과 뒤에서 삽입과 삭제가 좋다.  
Deque이 다른 자료구조와 가장 다른 점으로 앞과 뒤에서 삽입, 삭제가 좋다.
3. 중간에 데이터 삽입, 삭제가 용이하지 않다.  
데이터를 중간에 삽입하거나 삭제하는 것은 피해야 한다. 삽입과 삭제를 중간에 한다면 삽입하거나 삭제한 위치 앞뒤 데이터를 모두 이동해야 한다.

그림 6-3 데이터 g를 중간에 삽입하는 과정

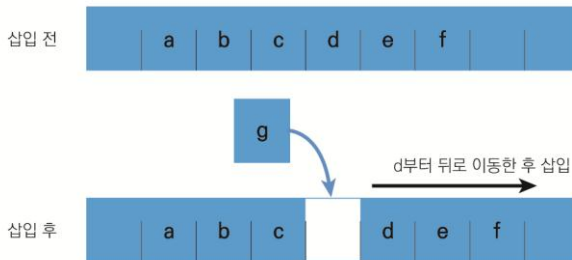
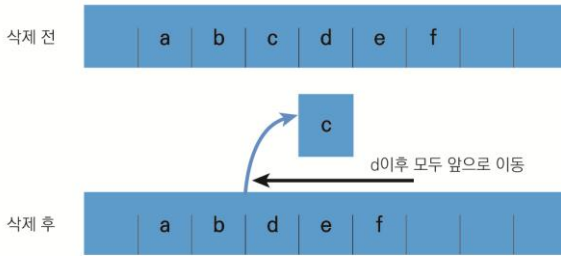


그림 6-4 데이터 c를 삭제하는 과정



4. 구현이 쉽지 않다.

Deque은 Stack과 Queue가 결합된 자료구조로 연결 리스트보다 구현하기가 더 어렵다.

5. 랜덤 접근이 가능하다.

연결 리스트처럼 리스트를 탐색하지 않고 원하는 요소에 바로 접근할 수 있다.

## 6.3 deque을 사용하는 경우

Deque의 특징을 고려할 때 다음과 같은 경우에 사용하면 좋다.

1. 앞과 뒤에서 삽입, 삭제를 한다.

이것이 deque을 사용하는 가장 큰 이유이다. 대부분 작업이 데이터를 앞이나 뒤에 삽입, 삭제를 한다면 STL 컨테이너 라이브러리 중에서 deque을 사용할 때 성능이 가장 좋다.

2. 저장할 데이터 개수가 가변적이다.

저장할 데이터 개수를 미리 알 수 없어도 deque은 크기가 동적으로 변하므로 유연하게 사용할 수 있다.

3. 검색을 거의 하지 않는다.

많은 데이터를 저장한다면 앞에서 여러 번 언급했듯이 map, set, hash\_map 중 하나를 선택해서 사용하는 편이 좋다.

4. 데이터 접근을 랜덤하게 하고 싶다.

vector와 같이 랜덤 접근이 가능하다. 사용하는 방법도 같다.

## 6.4 deque vs. vector

deque은 전체적으로 멤버 함수의 기능이나 사용방법이 vector와 거의 같다. vector는 삽입과 삭제를 뒤(back)에서만 해야 성능이 좋지만, deque은 삽입과 삭제를 앞과 뒤에서 해도 좋으며 앞뒤 삽입, 삭제 성능도 vector보다 좋다. 하지만, deque은 앞뒤에서 삽입, 삭제하는 것을 제외한 다른 위치에서의 삽입과 삭제는 vector보다 성능이 좋지 않다.

그림 6-5 deque과 vector의 차이

	deque	vector
크기 변경 가능	O	O
앞에 삽입, 삭제 용이	O	X
뒤에 삽입, 삭제 용이	O	O
중간 삽입, 삭제 용이	X	X
순차 접근 가능	O	O
랜덤 접근 가능	O	X

deque과 vector를 비교할 때 고려해야 되는 점은

- deque은 앞과 뒤에서 삽입과 삭제 성능이 vector보다 더 좋다.
- deque은 앞과 뒤 삽입, 삭제를 제외한 기능은 vector보다 성능이 좋지 못하다.

게임 서버에서 deque을 사용하는 경우

게임 서버는 클라이언트에서 보낸 패킷을 차례대로 처리한다. 서버에서 네트워크 데이터를 받는 함수에서 데이터를 받으면 패킷으로 만든 후 받은 순서대로 순차적으로 처리한다. 이렇게 순차적으로 저장한 패킷을 처리할 때는 deque이 가장 적합한 자료구조이다. 다만, 실제 현업에서는 이 부분에 STL의

deque을 사용하지 않는 경우가 종종 있다. 왜냐하면, 네트워크에서 데이터를 받아 패킷으로 만들어 저장하고, 그 패킷을 처리하는 부분은 게임 서버의 성능 면에서 가장 중요한 부분이므로 deque보다 더 빠르게 처리하기를 원한다. 따라서, 독자적인 자료구조를 만들어 사용한다(즉, 범용성보다는 성능을 우선 시한다).

## 6.5 deque 사용방법

deque을 사용하려면 deque 헤더 파일을 포함한다.

```
#include <deque>
```

deque 형식은 아래와 같다.

```
deque< 자료형 > 변수 이름
```

int를 사용하는 deque 선언은 아래와 같다.

```
deque< int > deque1;
```

deque도 동적 할당을 할 수 있다.

```
deque< 자료형 >* 변수 이름 = new deque< 자료형 >;
deque< int >* deque1 = new deque< int >;
```

### 6.5.1 deque의 주요 멤버들

deque 멤버 중 일반적으로 자주 사용하는 멤버들이다. vector에 없는 pop\_front와 push\_front가 있다는 것 빼고는 vector의 기능과 같다.

표 6-1 자주 사용하는 deque 멤버

멤버	설명
assign	특정 원소로 채운다
at	특정 위치의 원소의 참조를 리턴
back	마지막 원소의 참조를 리턴

begin	첫 번째 원소의 랜덤 접근 반복자를 리턴
clear	모든 원소를 삭제
empty	아무것도 없으면 true 리턴
End	마지막 원소 다음의(사용하지 않은 영역) 반복자를 리턴
erase	특정 위치의 원소나 지정 범위의 원소를 삭제
front	첫 번째 원소의 참조를 리턴
insert	특정 위치에 원소 삽입
pop_back	마지막 원소를 삭제
pop_front	첫 번째 원소를 삭제
push_back	마지막에 원소를 추가
push_front	제일 앞에 원소 추가
rbegin	역방향으로 첫 번째 원소의 반복자를 리턴
rend	역방향으로 마지막 원소 다음의 반복자를 리턴
reserve	지정된 크기의 저장 공간을 확보
size	원소의 개수를 리턴
swap	deque 의 두 원소를 서로 교환한다.

## 6.5.2 기본 사용 멤버

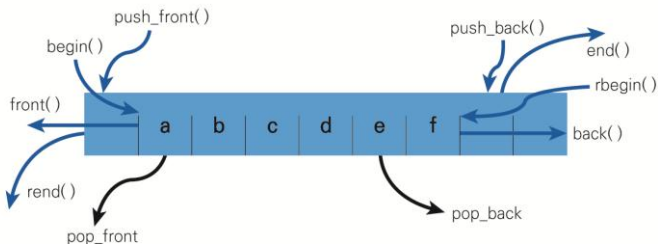
deque의 가장 기본적인(추가, 삭제, 접근 등) 사용법을 설명한다.

표 6-2 추가, 삭제, 접근 등에 사용하는 멤버들

begin	const_iterator begin() const; iterator begin();	첫 번째 원소의 랜덤 접근 반복자를 리턴
clear	void clear();	모든 원소를 삭제
empty	bool empty() const;	아무것도 없으면 true 리턴
end	iterator end(); const_iterator end() const;	마지막 원소 다음의(미 사용 영역) 반복자를 리턴
front	reference front(); const_reference front() const;	첫 번째 원소의 참조를 리턴
pop_back	void pop_back();	마지막 원소를 삭제
pop_front	void pop_front();	첫 번째 원소를 삭제

push_back	void push_back( const Type& _Val );	마지막에 원소를 추가
push_front	void push_front( const Type& _Val );	제일 앞에 원소 추가
rbegin	reverse_iterator rbegin( ); const_reverse_iterator rbegin( ) const;	역방향으로 첫 번째 원소의 반복자를 리턴
rend	const_reverse_iterator rend( ) const; reverse_iterator rend( );	역방향으로 마지막 원소 다음의 반복자를 리턴
size	size_type size() const;	원소의 개수를 리턴

그림 6-6 deque의 추가, 삭제, 접근 멤버들



## 추가

앞과 뒤에 추가를 할 수 있다. 보통 deque을 사용할 때는 뒤에 추가를 하고 앞에서는 삭제를 한다. 앞쪽 추가는 push\_front()를 사용한다.

```
deque< int > deque1;
deque1.push_front( 100 );
```

뒤에 추가할 때는 push\_back()을 사용한다.

```
deque< int > deque1;
deque1.push_back( 200 );
```



## 삭제

앞과 뒤에서 삭제 할 수 있다. 앞에서 삭제를 할 때는 `pop_front()`를 사용한다.

```
deque1.pop_front();
```

뒤에서 삭제를 할 때는 `pop_back()`를 사용한다.

```
deque1.pop_back();
```

## 접근

첫 번째 위치의 반복자를 얻을 때는 `begin()`을 사용한다.

```
deque< int >::iterator IterBegin = deque1.begin();  
cout << *IterBegin << endl;
```

반복자로 다른 원소에 접근을 할 때는 반복자에 „++“ 이나 „--“을 사용한다.

---

```
deque< int >::iterator IterPos = deque1.begin();  
// 다음 원소 위치로 이동  
++IterPos;  
// 이전 원소 위치로 이동  
--IterPos;
```

---

`end()`는 deque에 저장된 원소 중 마지막 다음 위치, 즉 사용하지 못하는 영역을 가리킵니다. 보통 반복문에서 컨테이너에 남은 원소가 있는지 조사할 때 주로 사용한다.

---

```
deque< int >::iterator IterEnd = deque1.end();  
for(deque< int >::iterator IterPos = deque1.begin; IterPos != IterEnd;  
++IterPos )  
{  
    .....  
}
```

---

첫 번째 위치에 있는 데이터를 얻을 때는 `front()`, 마지막 위치에 있는 데이터를 얻을 때는 `back()`을 사용한다.

```
int& FirstValue = deque1.front();  
int& LastValue = deque1.back();
```

`begin()`과 `end()`는 순방향으로 앞과 뒤를 가리키고, 역방향으로는 `rbegin()`과 `rend()`를 사용한다. 특정 위치에 있는 데이터를 얻을 때는 `at()`이나 배열 식 접근(`[]`)을 사용한다.

```
int& Value2 = deque1.at(1); // 두 번째 위치  
int Value3 = deque1[2];    // 세 번째 위치
```

## 모두 삭제

`clear()`를 사용하면 저장한 모든 데이터를 삭제한다.

```
deque1.clear();
```

## 데이터 저장 여부

`deque`에 저장한 데이터가 있는지 없는지는 `empty()`로 조사한다. 데이터가 있으면 `false`, 없다면 `true`를 리턴한다.

```
bool bEmpty = deque1.empty();
```

## 저장된 원소 개수 조사

```
size()로 deque에 저장된 데이터 개수를 조사한다.  
deque< int >::size_type TotalCount = deque1.size();
```

지금까지 설명한 `deque` 멤버들의 사용법을 보면 앞 장에서 설명한 `vector`와 같다는 것을 충분히 알 수 있을 것이다. `vector`를 공부하신 분들은 아주 쉽죠? ^^ 이후에 소개하는 내용도 `vector`에서 설명한 것과 같으니 편안하게 잘 따라오도록 한다.

### 6.5.3 deque 실습 예제

다음은 deque에서 가장 자주 사용하는 멤버들을 사용하는 전체 코드다.

#### [리스트 6-1]

---

```
#include <iostream>
#include <deque>

using namespace std;

// 서버/ 클라이언트간에 주고 받는 패킷
struct Packet
{
    unsigned short Index; // 패킷 인덱스
    unsigned short BodySize; // 패킷 보디(실제 데이터)의 크기
    char          acBodyData[100]; // 패킷의 데이터
};

int main()
{
    Packet Pkt1;
    Pkt1.Index = 1;    Pkt1.BodySize = 10;

    Packet Pkt2;
    Pkt2.Index = 2;    Pkt2.BodySize = 12;

    Packet Pkt3;
    Pkt3.Index = 3;    Pkt3.BodySize = 14;

    deque< Packet > ReceivePackets;

    // 뒤에 추가
    ReceivePackets.push_back( Pkt2 );
    ReceivePackets.push_back( Pkt3 );
```

---

---

```

// 앞에 추가
ReceivePackets.push_front( Pkt1 );

// 저장된 패킷 정보 출력
for( deque< Packet >::iterator iterPos = ReceivePackets.begin();
    iterPos != ReceivePackets.end();
    ++iterPos )
{
    cout << "패킷 인덱스: " << iterPos->Index << endl;
    cout << "패킷 바디 크기: " << iterPos->BodySize << endl;
}

cout << endl << "역방향으로 출력" << endl;
for( deque< Packet >::reverse_iterator iterPos = ReceivePackets.rbegin();
    iterPos != ReceivePackets.rend();
    ++iterPos )
{
    cout << "패킷 인덱스: " << iterPos->Index << endl;
    cout << "패킷 바디 크기: " << iterPos->BodySize << endl;
}

cout << endl << "배열 방식으로 접근" << endl;
// 저장된 총 패킷 수
int ReceivePacketCount = ReceivePackets.size();
cout << "총 패킷 수: " << ReceivePacketCount << endl;
for( int i = 0; i < ReceivePacketCount; ++i )
{
    cout << "패킷 인덱스: " << ReceivePackets[i].Index << endl;
    cout << "패킷 바디 크기: " << ReceivePackets[i].BodySize << endl;
}

// 첫 번째, 마지막 위치에 있는 패킷
Packet& FirstPacket = ReceivePackets.front();
cout << "첫 번째 패킷의 인덱스: " << FirstPacket.Index << endl;

Packet& LastPacket = ReceivePackets.back();

```

---

---

```
cout << "마지막 패킷의 인덱스: " << LastPacket.Index << endl;

// at을 사용하여 두 번째 패킷
Packet& PacketAt = ReceivePackets.at(1);
cout << "두 번째 패킷의 인덱스: " << PacketAt.Index << endl;

// 첫 번째 패킷 삭제
ReceivePackets.pop_front();
cout << "첫 번째 패킷의 인덱스: " << ReceivePackets[0].Index << endl;

// 마지막 패킷 삭제
ReceivePackets.pop_back();
LastPacket = ReceivePackets.back();
cout << "마지막 패킷의 인덱스: " << LastPacket.Index << endl;

// 모든 패킷을 삭제
if( false == ReceivePackets.empty() )
{
    cout << "모든 패킷을 삭제합니다." << endl;
    ReceivePackets.clear();
}
}
```

---

[리스트 6-1]의 실행 결과는 다음과 같다.

```
패킷 인덱스: 1
패킷 바디 크기: 10
패킷 인덱스: 2
패킷 바디 크기: 12
패킷 인덱스: 3
패킷 바디 크기: 14

역방향으로 출력
패킷 인덱스: 3
패킷 바디 크기: 14
패킷 인덱스: 2
패킷 바디 크기: 12
패킷 인덱스: 1
패킷 바디 크기: 10

배열 방식으로 접근
총 패킷 수: 3
패킷 인덱스: 1
패킷 바디 크기: 10
패킷 인덱스: 2
패킷 바디 크기: 12
패킷 인덱스: 3
패킷 바디 크기: 14
첫 번째 패킷의 인덱스: 1
마지막 패킷의 인덱스: 3
두 번째 패킷의 인덱스: 2
첫 번째 패킷의 인덱스: 2
마지막 패킷의 인덱스: 2
모든 패킷을 삭제합니다.
```

## insert 멤버

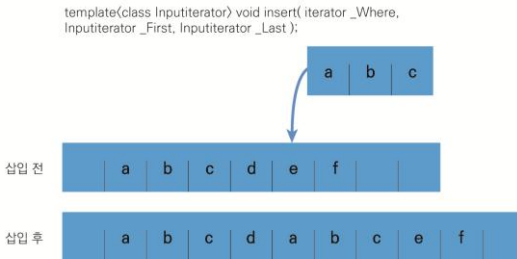
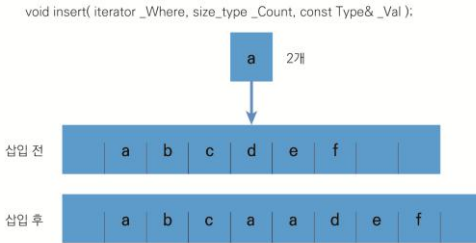
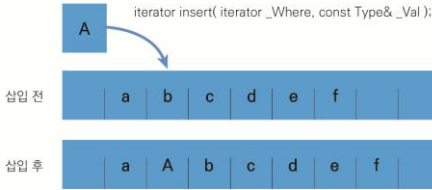
deque의 insert는 vector의 insert와 사용방법이 같다. 지정한 위치에 삽입, 지정한 위치에 지정한 개수만큼 삽입, 지정한 위치에 반복자 구간 안에 있는 것을 삽입한다. vector와 같이 insert는 삽입 되는 위치 이후에는 있는 모든 원소들이 뒤로 이동을 한다. 그리고 insert의 성능은 vector보다도 더 좋지 않다.

---

```
원형 : iterator insert( iterator _Where, const Type& _Val );
      void insert( iterator _Where, size_type _Count, const Type& _Val );
      template void insert( iterator _Where, InputIterator _First, InputIterator
        _Last );
```

---

그림 6-7 deque의 insert 처리 과정



지정한 위치에 데이터 삽입. 아래는 300을 첫 번째 위치에 삽입한다.

```
deque< int >::iterator iterInsertPos = deque1.begin();
deque1.insert( iterInsertPos, 300 );
```

지정한 위치에 데이터를 횟수만큼 삽입. 아래는 두 번째 위치에 10을 3번 추가한다.

---

```
iterInsertPos = deque1.begin();
++iterInsertPos;
deque1.insert( iterInsertPos, 3, 10 );
```

---

지정한 위치에 반복자의 시작과 끝 안에 있는 원소를 삽입. 아래는 두 번째 위치에 deque2의 처음과 끝에 있는 원소를 삽입한다.

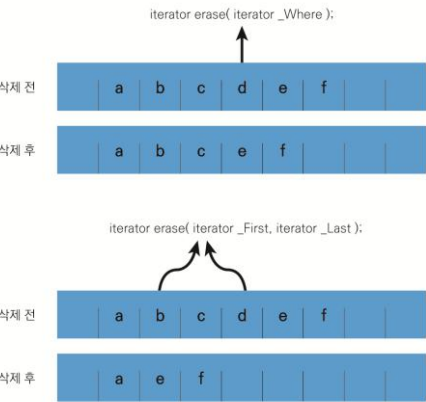
```
deque< int > deque2;  
deque2.push_back( 20 );  
deque2.push_back( 30 );  
deque2.push_back( 40 );  
  
iterInsertPos = deque1.begin();  
deque1.insert( ++iterInsertPos, deque2.begin(), deque2.end() );
```

erase 멤버

지정한 위치에 있는 원소를 삭제, 지정한 범위 안에 있는 원소를 삭제한다. vector와 사용법이 같고 또 erase를 하는 위치 이후의 모든 원소들이 앞으로 이동하는 것도 같다.

```
원형 : iterator erase( iterator _Where );  
       iterator erase( iterator _First, iterator _Last );
```

그림 6-8 deque의 erase 처리 과정





지정한 위치의 요소를 삭제. 아래는 첫 번째 요소를 삭제하는 코드다.

```
deque1.erase( deque1.begin() );
```

지정한 반복자 구간 안의 원소를 삭제한다. 아래는 deque1의 첫 번째 요소에서 마지막까지 모두 삭제한다.

```
deque1.erase( deque1.begin(), deque1.end() );
```

다음은 insert와 erase 사용 예다.

#### [리스트 6-2] Insert와 erase

---

```
#include <iostream>
#include <deque>

using namespace std;

int main()
{
    Packet Pkt1;
    Pkt1.Index = 1;    Pkt1.BodySize = 10;

    Packet Pkt2;
    Pkt2.Index = 2;    Pkt2.BodySize = 12;

    Packet Pkt3;
    Pkt3.Index = 3;    Pkt3.BodySize = 14;

    Packet Pkt4;
    Pkt4.Index = 4;    Pkt4.BodySize = 16;

    deque< Packet > ReceivePackets;
    ReceivePackets.push_back( Pkt1 );
    ReceivePackets.push_back( Pkt2 );
    ReceivePackets.push_back( Pkt3 );

    cout << "< insert >" << endl;

    // 첫 번째 위치에 Pkt3을 삽입
```

---

---

```

cout << "insert 1" << endl;
ReceivePackets.insert( ReceivePackets.begin(), Pkt3 );

for( deque< Packet >::iterator iterPos = ReceivePackets.begin();
    iterPos != ReceivePackets.end();
    ++iterPos )
{
    cout << "패킷 인덱스: " << iterPos->Index;
    cout << "          패킷 바디 크기: " << iterPos->BodySize << endl;
}

// 두 번째 위치에 Pkt4를 2개 삽입
cout << endl << "insert 2" << endl;
ReceivePackets.insert( ++ReceivePackets.begin(), 2, Pkt4 );
for( deque< Packet >::iterator iterPos = ReceivePackets.begin();
    iterPos != ReceivePackets.end();
    ++iterPos )
{
    cout << "패킷 인덱스: " << iterPos->Index;
    cout << "          패킷 바디 크기: " << iterPos->BodySize << endl;
}

deque< Packet > ReceivePackets2;
ReceivePackets2.push_back( Pkt3 );
ReceivePackets2.push_back( Pkt4 );
ReceivePackets2.push_back( Pkt1 );

// ReceivePackets2의 모든 것을 ReceivePackets의 첫 번째 위치에 삽입
cout << endl << "insert 3" << endl;
ReceivePackets.insert( ReceivePackets.begin(), ReceivePackets2.begin(),
                      ReceivePackets2.end() );
for( deque< Packet >::iterator iterPos = ReceivePackets.begin();
    iterPos != ReceivePackets.end();
    ++iterPos )
{
    cout << "패킷 인덱스: " << iterPos->Index;
    cout << "          패킷 바디 크기: " << iterPos->BodySize << endl;
}

```

---

---

```
cout << endl << "< erase >" << endl;
// 두 번째 원소를 삭제한다.
cout << "erase 1" << endl;
ReceivePackets.erase( ++ReceivePackets.begin() );
for( deque< Packet >::iterator iterPos = ReceivePackets.begin();
    iterPos != ReceivePackets.end();
    ++iterPos )
{
    cout << "패킷 인덱스: " << iterPos->Index;
    cout << "          패킷 바디 크기: " << iterPos->BodySize << endl;
}

// 두 번째 이후부터 모두 삭제한다.
cout << endl << "erase 2" << endl;
ReceivePackets.erase( ++ReceivePackets.begin(), ReceivePackets.end() );
for( deque< Packet >::iterator iterPos = ReceivePackets.begin();
    iterPos != ReceivePackets.end();
    ++iterPos )
{
    cout << "패킷 인덱스: " << iterPos->Index;
    cout << "          패킷 바디 크기: " << iterPos->BodySize << endl;
}
}
```

---

[리스트 6-2]의 실행 결과는 다음과 같다.

```

< insert >
insert 1
패킷 인덱스: 3      패킷 바디 크기: 14
패킷 인덱스: 1      패킷 바디 크기: 10
패킷 인덱스: 2      패킷 바디 크기: 12
패킷 인덱스: 3      패킷 바디 크기: 14

insert 2
패킷 인덱스: 3      패킷 바디 크기: 14
패킷 인덱스: 4      패킷 바디 크기: 16
패킷 인덱스: 4      패킷 바디 크기: 16
패킷 인덱스: 1      패킷 바디 크기: 10
패킷 인덱스: 2      패킷 바디 크기: 12
패킷 인덱스: 3      패킷 바디 크기: 14

insert 3
패킷 인덱스: 3      패킷 바디 크기: 14
패킷 인덱스: 4      패킷 바디 크기: 16
패킷 인덱스: 1      패킷 바디 크기: 10
패킷 인덱스: 3      패킷 바디 크기: 14
패킷 인덱스: 4      패킷 바디 크기: 16
패킷 인덱스: 4      패킷 바디 크기: 16
패킷 인덱스: 4      패킷 바디 크기: 16
패킷 인덱스: 1      패킷 바디 크기: 10
패킷 인덱스: 2      패킷 바디 크기: 12
패킷 인덱스: 3      패킷 바디 크기: 14

< erase >
erase 1
패킷 인덱스: 3      패킷 바디 크기: 14
패킷 인덱스: 1      패킷 바디 크기: 10
패킷 인덱스: 3      패킷 바디 크기: 14
패킷 인덱스: 4      패킷 바디 크기: 16
패킷 인덱스: 4      패킷 바디 크기: 16
패킷 인덱스: 1      패킷 바디 크기: 10
패킷 인덱스: 2      패킷 바디 크기: 12
패킷 인덱스: 3      패킷 바디 크기: 14

erase 2
패킷 인덱스: 3      패킷 바디 크기: 14

```

## assign

vector의 assign과 같이 deque의 assign도 컨테이너를 특정 데이터로 채울 때 사용한다. 특정 값이나 다른 deque의 특정 영역(반복자로 가리키는)에 있는 데이터로 채울 수 있다. assign을 사용하는 deque에 데이터가 있다면 이를 덮어쓰면서 채운다.

---

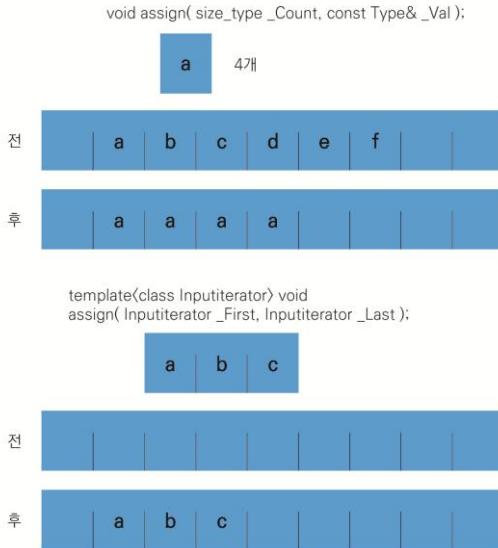
```

원형 : void assign( size_type _Count, const Type& _Val );
template void assign( InputIterator _First, InputIterator _Last );

```

---

그림 6-9 deque의 assign



지정 데이터를 지정 개수만큼 채웁니다. 숫자 7를 7개 채운다.

```
deque1.assign( 7, 7 );
```

다른 deque의 반복자로 지정한 영역으로 채운다.

```
deque1.assign( deque2.begin(), deque2.end() );
```

## swap

deque1과 deque2가 있을 때 두 deque에 저장한 데이터를 서로 맞바꿀 때 사용한다. swap 원형은 아래와 같다.

---

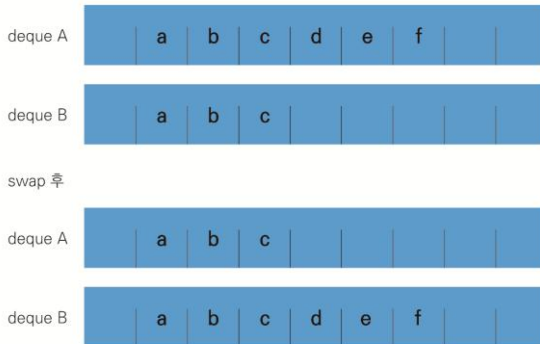
```
원형 : void swap( deque& _Right );
friend void swap( deque& _Left, deque& _Right )
template void swap( deque< Type, Allocator>& _Left, deque< Type, Allocator>&
```

\_Right );

---

deque A와 B를 swap하는 모습을 그림으로 나타내면 아래와 같다.

그림 6-10 deque의 swap



다음은 deque1과 deque2를 swap하는 방법을 두 가지로 나타낸 코드다.

```
deque1.swap( deque2 );  
swap(deque1, deque2);
```

[리스트 6-3] assign, swap

---

```
#include <iostream>  
#include <deque>  
  
using namespace std;  
  
int main()  
{  
    deque< int > deque1;  
  
    cout << "assign 1" << endl;  
    deque1.assign( 7, 7 );  
    for( deque< int >::iterator iterPos = deque1.begin();  
        iterPos != deque1.end();
```

---

---

```

        ++iterPos )
    {
        cout << "deque 1 : " << *iterPos << endl;
    }

    cout << endl << "assign 2" << endl;
    deque< int > deque2;
    deque2.assign( deque1.begin(), deque1.end() );
    for( deque< int >::iterator iterPos = deque2.begin();
        iterPos != deque2.end();
        ++iterPos )
    {
        cout << "deque 2 : " << *iterPos << endl;
    }

    // swap
    deque< int > deque3;
    deque3.push_back(10);
    deque3.push_back(20);
    deque3.push_back(30);

    cout << endl << "swap" << endl;
    deque3.swap( deque1 );
    for( deque< int >::iterator iterPos = deque3.begin();
        iterPos != deque3.end();
        ++iterPos )
    {
        cout << "deque 3 : " << *iterPos << endl;
    }

    for( deque< int >::iterator iterPos = deque1.begin();
        iterPos != deque1.end();
        ++iterPos )
    {
        cout << "deque 1 : " << *iterPos << endl;
    }
}

```

---

[리스트 6-3]의 실행 결과는 다음과 같다.

```
assign 1
deque 1 : 7
deque 1 : 7
deque 1 : 7
deque 1 : 7
deque 1 : 7
deque 1 : 7
deque 1 : 7

assign 2
deque 2 : 7
deque 2 : 7
deque 2 : 7
deque 2 : 7
deque 2 : 7
deque 2 : 7
deque 2 : 7

swap
deque 3 : 7
deque 3 : 7
deque 3 : 7
deque 3 : 7
deque 3 : 7
deque 3 : 7
deque 3 : 7
deque 1 : 10
deque 1 : 20
deque 1 : 30
```

deque에서 자주 사용하는 멤버를 중심으로 설명했다. 여기에서 설명하지 않은 나머지 deque 멤버의 사용법은 여기(<http://msdn.microsoft.com/en-us/library/8tk0b6f0.aspx>)에서 참고한다. 앞에서 여러 번 언급했듯이 deque 사용법은 5장의 vector와 거의 같다. 위 예제에서 deque를 vector로 바꾸어도 push\_front()를 제외하고는 문제없이 컴파일할 수 있다. deque와 vector는 사용법은 비슷하나 자료구조는 완전히 다르다. 앞과 끝에 데이터를 추가, 삭제하는 일이 대부분이라면 deque가 vector보다 좋지만, 그렇지 않다면 vector를 사용하는 쪽이 훨씬 더 좋다. STL 컨테이너는 사용이 어렵다기보다는 언제 어떤 STL 컨테이너를 사용해야 알맞은지 선택하기가 어렵다. STL 컨테이너를 올바른 장소에 사용하려면 컨테이너의 자료구조가 무엇인지 확실하게 알고 있어야 한다. 만약 자료구조를 잘 모른다면 꼭 자료구조 공부를 STL 공부보다 먼저 해야 한다.



## 6.6 과제

1. deque를 사용하여 FIFO, LIFO 방식의 스택을 만들어본다.
2. deque를 사용하여 Undo, Redo 구현해 보라.

간단한 예제로 deque의 원리만 익혀보는 과제다. Deque에 숫자를 저장하고 Undo를 하면 가장 마지막에 넣은 데이터를 빼고 Redo를 하면 뺀 데이터를 다시 넣는다. deque를 2개 사용하면 횟수 제한이 없는 Undo, Redo를 구현할 수 있다.

## 7 해시 맵(hash\_map)

이 글을 보는 독자는 대부분 아직 STL을 잘 모를 거로 생각한다. 필자가 일하고 있는 게임업체는 주력 언어가 C++이다. 그래서 취업 사이트에 올라온 프로그래머 채용 공고를 보면 필수 조건에 거의 대부분이 C++와 STL 사용 가능성이 들어가 있다. 게임업체뿐 아니라 C++을 사용하여 프로그래밍하는 곳이라면 대부분 C++과 STL을 사용하여 프로그램을 만들 수 있는 실력이 있어야 한다.

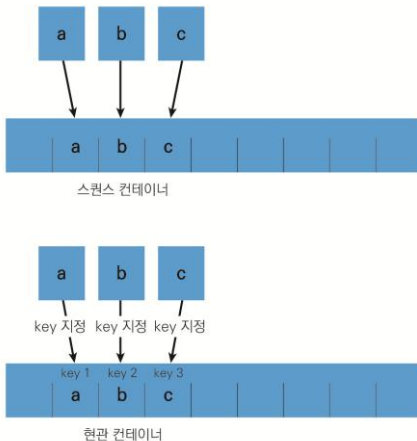
C++ 언어를 배우고 사용하는 프로그래머라면 STL을 배우면 좋고, 특히 게임 프로그래머가 되기를 원하는 사람은 STL을 꼭 사용할 줄 알아야 한다.

### 7.1 시퀀스 컨테이너와 연관 컨테이너

6장까지 STL의 컨테이너에 대해서 설명했다. STL 컨테이너는 크게 시퀀스 컨테이너와 연관 컨테이너로 나눈다. 시퀀스 컨테이너는 vector, list, deque와 같이 순서 있게 자료를 보관한다.

연관 컨테이너는 어떠한 Key와 짝을 이루어 자료를 보관한다. 그래서 자료를 넣고, 빼고, 찾을 때는 Key가 필요하다.

그림 7-1 시퀀스 컨테이너와 연관 컨테이너



시퀀스 컨테이너는 많지 않은 자료를 보관하고 검색 속도가 중요한 경우에 사용하고, 연관 컨테이너는 대량의 자료를 보관하고 검색을 빠르게 하고 싶을 때 사용한다. 필자가 만드는 온라인 게임 서버에서는 보통 접속한 유저들의 정보를 보관할 때 가장 많이 사용한다.

## 7.2 연관 컨테이너로는 무엇이 있을까?

연관 컨테이너로 `map`, `set`, `hash_map`, `hash_set`이 있다. 이것들은 Key로 사용하는 값이 중복되지 않은 때 사용한다. 만약 중복되는 key를 사용할 때는 컨테이너의 앞에 'multi'를 붙인 `multi_map`, `multi_set`, `hash_multimap`, `hash_multiset`을 사용한다. Key의 중복 허용 여부만 다를 뿐 사용방법은 같다.

### 7.2.1 `map`, `set` 과 `hash_map`, `hash_set`의 차이는?

가장 쉽게 알 수 있는 큰 차이는 이름 앞에 'hash'라는 단어가 있느냐 없느냐의 차이이다. 그렇다. 'hash'라는 단어가 정말 큰 차이이다.

`map`과 `set` 컨테이너는 자료를 정렬하여 저장한다. 그래서 반복자로 저장된 데이터를 순회할 때 넣은 순서로 순회하지 않고 정렬된 순서대로 순회한다. `hash_map`, `hash_set`은 정렬 하지 않으며 자료를 저장한다. 또 hash라는 자료구조를 사용함으로 검색 속도가 `map`, `set`에 비해 빠르다.

`map`, `set`과 `hash_map`, `hash_set` 중 어느 것을 사용할지 선택할 때는

`map`, `set`의 사용하는 경우 : 정렬된 상태로 자료 저장을 하고 싶을 때.  
`hash_map`, `hash_set` : 정렬이 필요 없으며 빠른 검색을 원할 때.

를 가장 큰 조건으로 보면 좋다.

### 7.2.2 `hash_map`, `hash_set`은 표준은 아니다.

위에 열거한 연관 컨테이너 중 `map`과 `set`은 STL 표준 컨테이너지만 `hash_map`, `hash_set`은 표준이 아니다. 그래서 보통 STL 관련 책을 보시면 `hash_map`과 `hash_set`에 대한 설명은 없다. `hash_map`, `hash_set`을 쓰려면 라이브러리를 설치해야 할까? 그럴 필요는 없다. STL 표준은 아니지만 오래되지

많은 C++ 컴파일러에서는 대부분 지원한다. 윈도우에서는 Visual Studio.NET 이후의 모든 버전에서 지원한다.

STL 표준도 아닌 `hash_map`을 설명하려는 이유는 대부분 C++ 컴파일러에서 지원하고, 새로운 C++ 표준에서는 정식으로 STL에 들어갈 예정이며 현업에서 프로그래밍 할 때 아주 유용하게 사용하는 컨테이너이기 때문이다.

#### [참고]

2011년에 C++11이라고 부르는 C++표준이 나왔다. 표준이 공표되기 전인 2007년에 TR1이라는 이름으로 일부 공개했다. TR1에서는 `hash_map`, `hash_set`과 거의 같은 컨테이너인 `unordered_map`, `unordered_set`이 준비되어 있다. `hash_map`, `hash_set`과 이름만 다를 뿐 컨테이너의 자료구조나 사용 방법이 거의 같다.

그래서 `hash_map`, `hash_set` 사용법을 익히며 자동으로 `unordered_map`, `unordered_set`도 익히게 된다.

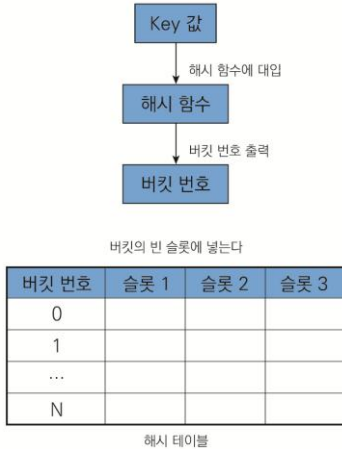
**NOTE** 이 책에서 말하는 '설계'는 '소프트웨어 설계'를 의미한다. 다시 말해서 시각 디자인, 사용자 인터페이스 디자인 같은 시각적인 디자인을 의미하는 것이 아니다.

## 7.3 `hash_map`의 자료구조

`hash_map`의 자료구조는 '해시 테이블'이다. 다음 그림 7-2에 나와 있듯이 해시 테이블에 자료를 저장할 때는 Key 값을 해시 함수에 대입하여 버킷 번호가 나오면 그 버킷의 빈 슬롯에 자료를 저장한다.

Key 값을 해시 함수에 입력하여 나오는 버킷 번호에 자료를 넣으므로 많은 자료를 저장해도 삽입, 삭제, 검색 속도가 거의 일정하다.

그림 7-2 해시 테이블에 자료 넣기



해시 테이블에 대한 설명은 간단하지 않고 이 글에서는 `hash_map` 사용법을 간단하게 설명한다. 좀 더 자세한 사항은 아래 참고를 확인한다.

#### 참고 해시 테이블 설명

1. 좋은 프로그램을 만드는 핵심 원리 25가지(한빛미디어)

## 7.4 `hash_map`을 사용할 때와 사용하지 않을 때

6장까지 설명했던 STL 컨테이너의 장단점은 컨테이너의 자료구조를 보면 알 수 있다. `hash_map`은 해시 테이블을 자료구조로 사용하므로 해시 테이블에 대해 알면 장단점을 파악할 수 있다. 해시 테이블은 많은 자료를 저장하고 있어도 검색이 빠르다. 그러나 저장한 자료가 적을 때는 메모리 낭비와 검색 시 오버헤드가 생긴다.

Key 값을 해시 함수에 넣어 알맞은 버킷 번호를 알아 내는 것은 마법 같은 것이 아니다. 그러므로 `hash_map`은 해시 테이블을 사용하므로 검색이 빠르다라는 것만 생각하고 무분별하게 `hash_map`을 사용하면 안된다. 컨테이너에 추가나 삭제를 하는 것은 `list`나 `vector`, `deque`가 `hash_map`보다 빠르다. 또 적은

요소를 저장하고 검색할 때는 vector나 list가 훨씬 빠를 수 있다. 수천의 자료를 저장하여 검색을 하는 경우에 hash\_map을 사용하는 것이 좋다.

hash\_map을 사용하는 경우

1. 많은 자료를 저장하고, 검색 속도가 빨라야 한다.
2. 너무 빈번하게 자료를 삽입, 삭제 하지 않는다.

## 7.5 hash\_map 사용방법

hash\_map을 STL의 다른 컨테이너와 같이 사용하려면 먼저 헤더 파일과 namespace를 선언해야 한다. 그러나 여기서 주의할 점은 앞서 이야기했듯이 hash\_map은 표준이 아니므로 표준 STL의 namespace와 다른 이름을 사용한다. 따라서, namespace 선언할 때 실수하지 않게 조심해야 한다.

hash\_map 헤더 파일을 포함한다.

```
#include <hash_map>
```

hash\_map이 속한 namespace는 표준 STL과 다른 'stdext'이다.

```
using namespace stdext;
```

hash\_map 선언은 아래와 같다.

```
hash_map< Key 자료형, Value 자료형 > 변수 이름
```

위에서는 Value는 저장할 데이터이고, Key는 Value와 가리키는 데이터다. Key는 int, Value는 float를 사용한다면 아래와 같다.

```
hash_map< int, float > hash1;
```

다른 컨테이너와 같이 동적 할당을 할 수 있다.

```
hash_map< key 자료형, Value 자료형 >* 변수 이름 = new hash_map< key 자료형, Value 자료형 >;
```

```
hash_map< int, float >* hash1 = new hash_map< int, float >;
```

hash\_map은 Key와 Value가 짝을 이뤄야 하므로 hash\_map을 처음 보는 분들은 이전의 시퀀스 컨테이너와 다르게 좀 복잡하게 보일 것이다. 그러나 사용이 어려운 것은 아니니 잘 따라오도록 한다.

## 7.5.1 hash\_map의 주요 멤버들

표 7-1 자주 사용하는 hash\_map 멤버

멤버	설명
begin	첫 번째 원소의 랜덤 접근 반복자를 리턴
clear	저장한 모든 원소를 삭제
empty	저장한 요소가 없으면 true 리턴
end	마지막 원소 다음의(사용하지 않는 영역) 반복자를 리턴
erase	특정 위치의 원소나 지정 범위의 원소들을 삭제
find	Key와 연관된 원소의 반복자 리턴
insert	원소 추가
lower_bound	지정한 Key의 요소가 있다면 해당 위치의 반복자를 리턴
rbegin	역방향으로 첫 번째 원소의 반복자를 리턴
rend	역방향으로 마지막 원소 다음의 반복자를 리턴
size	원소의 개수를 리턴
upper_bound	지정한 Key 요소가 있다면 해당 위치 다음 위치의 반복자 리턴

hash\_map 컨테이너를 사용할 때는 거의 대부분 추가, 삭제, 검색 이렇게 3가지를 사용한다. 핵심 기능인 만큼 아래에 좀 더 자세하게 설명하고, 다른 컨테이너는 앞서 설명한 것과 사용방법이 같으므로 예제 코드로 보여준다.

## 7.5.2 추가

### insert

hash\_map에서는 자료를 추가할 때 insert를 사용한다.

원형 :

```
pair<iterator, bool> insert( const value_type& _Val );
iterator insert( iterator _Where, const value_type& _Val );
```

---

```
template<class InputIterator> void insert( InputIterator _First, InputIterator
_Last );
```

---

insert를 사용하는 세 가지 방법 중 첫 번째 방식으로 Key 타입은 int, Value 타입은 float를 추가한다면

---

```
hash_map<int, float> hashmap1, hashmap2;
// Key는 10, Value는 45.6f를 추가.
hashmap1.insert(hash_map<int, float>::value_type(10, 45.6f));
```

---

두 번째 방식으로는 특정 위치에 추가할 수 있다.

---

```
// 첫 번째 위치에 key 11, Value 50.2f를 추가
hashmap1.insert(hashmap1.begin(), hash_map<int, float>::value_type(11, 50.2f));
```

---

세 번째 방식으로는 지정한 반복자 구간에 있는 것들을 추가한다.

---

```
// hashmap1의 모든 요소를 hashmap2에 추가.
hashmap2.insert( hashmap1.begin(), hashmap1.end() );
```

---

## 7.5.3 삭제

### erase

---

원형 :

```
iterator erase( iterator _Where );
iterator erase( iterator _First, iterator _Last );
size_type erase( const key_type& _Key );
```

---

첫 번째 방식은 특정 위치에 있는 요소를 삭제한다.

---

```
// 첫 번째 위치의 요소 삭제.
hashmap1.erase( hashmap1.begin() );
```

---



두 번째 방식은 지정한 구역에 있는 요소들을 삭제한다.

---

```
// hashmap1의 처음과 마지막에 있는 모든 요소 삭제
hashmap1.erase( hashmap1.begin(), hashmap1.end() );
```

---

세 번째 방식은 지정한 키와 같은 요소를 삭제한다.

---

```
// Key가 11인 요소 삭제.
hashmap1.erase( 11 );
```

---

첫 번째와 두 번째 방식의 리턴 값으로는 삭제된 요소의 다음의 것을 가리키는 반복자이며 세 번째 방식은 삭제된 개수를 리턴한다.

## 7.5.4 검색

hahs\_map에서 검색은 Key를 사용하여 같은 Key를 가지고 있는 요소를 찾는다. Key와 같은 요소를 찾으면 그 요소의 반복자를 리턴하고, 찾지 못한 경우에는 end()를 가리키는 반복자를 리턴한다.

---

원형 :

```
iterator find( const Key& _Key );
const_iterator find( const Key& _Key ) const;
```

---

방식은 두 가지지만 사용법은 같다. 차이는 리턴된 반복자가 const냐 아니냐의 차이다. 참고로 첫 번째 방식은 const가 아니므로 찾은 요소의 Value를 변경할 수 있다(참고로 Key는 변경 할 수 없다). 두 번째 방식은 Value도 변경할 수 없다.

---

```
// Key가 10인 요소 찾기.
hash_map<int, float>::Iterator FindIter = hashmap1.find( 10 );
// 찾았다면 Value를 290.44로 변경
If( FindIter != hashmap1.end() )
{
    FindIter->second = 290.44f;
}
```

---

begin, clear, count, empty, end, rbegin, rend, size는 앞서 말 했듯이 다른 컨테이너와 사용방법이 비슷하므로 아래 예제 코드를 통해서 사용법을 보여 준다.

#### [리스트 7-1] hash\_map을 사용한 유저 관리

---

```
#include <iostream>
#include <hash_map>
using namespace std;
using namespace stdext;

// 게임 캐릭터
struct GameCharacter
{
    // 아래의 인자를 가지는 생성자를 정의한 경우는
    // 꼭 기본 생성자를 정의해야 컨테이너에서 사용할 수 있다.
    GameCharacter() { }

    GameCharacter( int CharCd, int Level, int Money )
    {
        _CharCd = CharCd;
        _Level = Level;
        _Money = Money;
    }
    int _CharCd;    // 캐릭터 코드
    int _Level;     // 레벨
    int _Money;     // 돈
};

void main()
{
    hash_map<int, GameCharacter> Characters;

    GameCharacter Character1(12, 7, 1000 );
    Characters.insert(hash_map<int, GameCharacter>::value_type(12, Character1));

    GameCharacter Character2(15, 20, 111000 );
    Characters.insert(hash_map<int, GameCharacter>::value_type(15, Character2));
```

---

---

```

GameCharacter Character3(200, 34, 3345000 );
Characters.insert(hash_map<int, GameCharacter>::value_type(200, Character3));

// iterator와 begin, end 사용
// 저장한 요소를 정방향으로 순회
hash_map<int, GameCharacter>::iterator Iter1;
for( Iter1 = Characters.begin(); Iter1 != Characters.end(); ++Iter1 )
{
    cout << "캐릭터 코드 : " << Iter1->second._CharCd << " |   레벨 : " <<
        Iter1->second._Level << " |   가진 돈 : " << Iter1->second._Money << endl;
}
cout << endl;

// rbegin, rend 사용
// 저장한 요소의 역방향으로 순회
hash_map<int, GameCharacter>::reverse_iterator RIter;
for( RIter = Characters.rbegin(); RIter != Characters.rend(); ++RIter )
{
    cout << "캐릭터 코드 : " << RIter->second._CharCd << " |   레벨 : " <<
        RIter->second._Level << " |   가진 돈 : " << RIter->second._Money << endl;
}
cout << endl << endl;

// Characters에 저장한 요소 수
int CharacterCount = Characters.size();

// 검색.
// 캐릭터 코드 15인 캐릭터를 찾는다.
hash_map<int, GameCharacter>::iterator FindIter = Characters.find(15);
// 찾지 못했다면 FindIter은 end 위치의 반복자가 리턴된다.
if( Characters.end() == FindIter )
{
    cout << "캐릭터 코드가 20인 캐릭터가 없습니다" << endl;
}
else
{
    cout << "캐릭터 코드가 15인 캐릭터를 찾았습니다." << endl;
    cout << "캐릭터 코드 : " << FindIter->second._CharCd << " |   레벨 : " <<
        FindIter->second._Level << " |   가진 돈 : " << FindIter->second._Money << endl;
}

```

---

---

```

    }
    cout << endl;

    for( Iter1 = Characters.begin(); Iter1 != Characters.end(); ++Iter1 )
    {
        cout << "캐릭터 코드 : " << Iter1->second._CharCd << " |   레벨 : " << Iter1-
            >second._Level << " |   가진 돈 : " << Iter1->second._Money << endl;
    }
    cout << endl << endl;

    // 삭제
    // 캐릭터 코드가 15인 캐릭터를 삭제한다.
    Characters.erase( 15 );
    for( Iter1 = Characters.begin(); Iter1 != Characters.end(); ++Iter1 )
    {
        cout << "캐릭터 코드 : " << Iter1->second._CharCd << " |   레벨 : " <<
            Iter1->second._Level << " |   가진 돈 : " << Iter1->second._Money << endl;
    }
    cout << endl << endl;

    // 모든 캐릭터를 삭제한다.
    Characters.erase( Characters.begin(), Characters.end() );

    // Characters 공백 조사
    if( Characters.empty() )
    {
        cout << "Characters는 비어 있습니다." << endl;
    }
}

```

---

[리스트 7-1]의 결과는 다음과 같다.

```
캐릭터 코드 : 12 : 레벨 : 7! 가진 돈 : 1000
캐릭터 코드 : 15 : 레벨 : 20! 가진 돈 : 111000
캐릭터 코드 : 200 : 레벨 : 34! 가진 돈 : 3345000

캐릭터 코드 : 200 : 레벨 : 34! 가진 돈 : 3345000
캐릭터 코드 : 15 : 레벨 : 20! 가진 돈 : 111000
캐릭터 코드 : 12 : 레벨 : 7! 가진 돈 : 1000

캐릭터 코드가 15인 캐릭터를 찾았습니다.
캐릭터 코드 : 15 : 레벨 : 20! 가진 돈 : 111000

캐릭터 코드 : 12 : 레벨 : 7! 가진 돈 : 1000
캐릭터 코드 : 15 : 레벨 : 20! 가진 돈 : 111000
캐릭터 코드 : 200 : 레벨 : 34! 가진 돈 : 3345000

캐릭터 코드 : 12 : 레벨 : 7! 가진 돈 : 1000
캐릭터 코드 : 200 : 레벨 : 34! 가진 돈 : 3345000

Characters는 비어 있습니다.
```

## 7.5.5 lower\_bound와 upper\_bound

hash\_map에 저장한 요소 중에서 Key 값으로 해당 요소의 시작 위치를 얻을 때 사용하는 멤버들이다. Key 값의 비교는 크기가 아닌 저장 되어 있는 요소의 순서이다. 23, 4, 5, 18, 14, 30 이라는 순서로 Key 값을 가진 요소가 저장되어 있으며 Key 값 18과 같거나 큰 것을 찾으면 18, 14, 30이 된다.

lower\_bound

Key가 있다면 해당 위치의 반복자를 리턴한다.

---

원형 :

```
iterator lower_bound( const Key& _Key );
const_iterator lower_bound( const Key& _Key ) const;
```

upper\_bound

---

Key가 있다면 그 요소 다음 위치의 반복자를 리턴한다.

---

원형 :

```
iterator lower_bound( const Key& _Key );  
const_iterator lower_bound( const Key& _Key ) const;
```

---

lower\_bound와 upper\_bound는 hahs\_map에 저장된 요소를 일부분씩 나누어 처리를 할 때 유용하다. 예를 들면 hash\_map에 3,000개의 게임 캐릭터 정보를 저장되어 있으며 이것을 100개씩 나누어 처리하고 싶을 때 사용하면 좋다.

다음은 lower\_bound와 upper\_bound 사용하는 예다.

#### [리스트 7-2] lower\_bound, upper\_bound

---

```
#include <iostream>  
#include <hash_map>  
using namespace std;  
using namespace stdext;  
  
// 게임 캐릭터  
struct GameCharacter  
{  
    GameCharacter() { }  
  
    GameCharacter( int CharCd, int Level, int Money )  
    {  
        _CharCd = CharCd;  
        _Level = Level;  
        _Money = Money;  
    }  
    int _CharCd;    // 캐릭터코드  
    int _Level;     // 레벨  
    int _Money;     // 돈  
};  
  
void main()  
{  
    hash_map<int, GameCharacter> Characters;  
  
    GameCharacter Character1(12, 7, 1000 );
```

---

---

```

Characters.insert(hash_map<int, GameCharacter>::value_type(12, Character1));

GameCharacter Character2(15, 20, 111000 );
Characters.insert(hash_map<int, GameCharacter>::value_type(15, Character2));

GameCharacter Character3(7, 34, 3345000 );
Characters.insert(hash_map<int, GameCharacter>::value_type(7, Character3));

GameCharacter Character4(14, 12, 112200 );
Characters.insert(hash_map<int, GameCharacter>::value_type(14, Character4));

GameCharacter Character5(25, 3, 5000 );
Characters.insert(hash_map<int, GameCharacter>::value_type(25, Character5));

hash_map<int, GameCharacter>::iterator Iter1;
cout << "저장한 캐릭터 리스트" << endl;
for( Iter1 = Characters.begin(); Iter1 != Characters.end(); ++Iter1 )
{
    cout << "캐릭터 코드 : " << Iter1->second._CharCd << " |   레벨 : " <<
        Iter1->second._Level << " |   가진 돈 : " << Iter1->second._Money << endl;
}
cout << endl;

cout << "lower_bound(14)" << endl;
hash_map<int, GameCharacter>::iterator Iter = Characters.lower_bound(14);
while( Iter != Characters.end() )
{
    cout << "캐릭터 코드 : " << Iter->second._CharCd << " |   레벨 : " <<
        Iter->second._Level << " |   가진 돈 : " << Iter->second._Money << endl;

    ++Iter;
}
cout << endl;

cout << "upper_bound(7)" << endl;
Iter = Characters.upper_bound(7);
while( Iter != Characters.end() )
{
    cout << "캐릭터 코드 : " << Iter->second._CharCd << " |   레벨 : " << Iter-

```

```

>second._Level << "| 가진 돈 : " << Iter->second._Money << endl;

++Iter;
}
}

```

[리스트 7-2]의 실행 결과는 다음과 같다.

```

저장한 캐릭터 리스트
캐릭터 코드 : 12 ! 레벨 : 7! 가진 돈 : 1000
캐릭터 코드 : 7 ! 레벨 : 34! 가진 돈 : 3345000
캐릭터 코드 : 15 ! 레벨 : 20! 가진 돈 : 111000
캐릭터 코드 : 14 ! 레벨 : 12! 가진 돈 : 112200
캐릭터 코드 : 25 ! 레벨 : 3! 가진 돈 : 5000

lower_bound(14)
캐릭터 코드 : 14 ! 레벨 : 12! 가진 돈 : 112200
캐릭터 코드 : 25 ! 레벨 : 3! 가진 돈 : 5000

upper_bound(7)
캐릭터 코드 : 15 ! 레벨 : 20! 가진 돈 : 111000
캐릭터 코드 : 14 ! 레벨 : 12! 가진 돈 : 112200
캐릭터 코드 : 25 ! 레벨 : 3! 가진 돈 : 5000

```

이것으로 hash\_map의 주요 사용법에 대한 설명이 끝났다. 위에 설명한 것들만 알고 있으면 hash\_map을 사용하는데 문제가 없을 것이다. 위에 설명한 것들 이외의 hash\_map의 멤버들에 대해서 알고 싶으면 다음 마이크로소프트의 MSDN 사이트를 참고한다.

[http://msdn.microsoft.com/en-us/library/h80zf4bx\(VS.80\).aspx](http://msdn.microsoft.com/en-us/library/h80zf4bx(VS.80).aspx)

**참고** Visual C++의 hash\_map의 성능에 대해서 Visual C++에 있는 hash\_map은 다른 컴파일에서 구현한 것보다 꽤 느리다라는 말이 있다. 관련 글은 다음 웹 사이트를 참고하였다. 얼마 나 느린지 테스트했다.

- 참고 웹 사이트 : <http://minjang.egloos.com/1983788>

필자가 조사한 것은 Windows 플랫폼에서 VC++에서 제공한 라이브러리로 테스트한 것이다. 결과를 보면 hash\_map이 map보다 빠르지도 않고 특히 hash\_map과 같은 자료구조를 사용하는 컨테이너로 마이크로소프트사에서 만든 CAtlMap에 비해 속도가 아주 느리다.



성능이 중요한 곳에 `hash_map`을 사용한다면 VC++에 있는 것을 사용하지 말고 자체적으로 잘 만들어진 `hash` 함수를 사용하거나 C++ 오픈 소스 라이브러리인 `boost`에 있는 `unordered_map`을 사용하는 것이 좋을 것 같다. Windows 플랫폼에서만 사용한다면 `CAtlMap`을 사용하는 것도 좋다.

## 8 맵(map)

이 장에서는 7장에서 설명한 hash\_map과 같은 연관 컨테이너 중의 하나인 map에 대해서 설명한다. 사용법이 hash\_map과 대부분 비슷해서 앞으로 할 이야기가 별로 어렵지 않을 것이다.

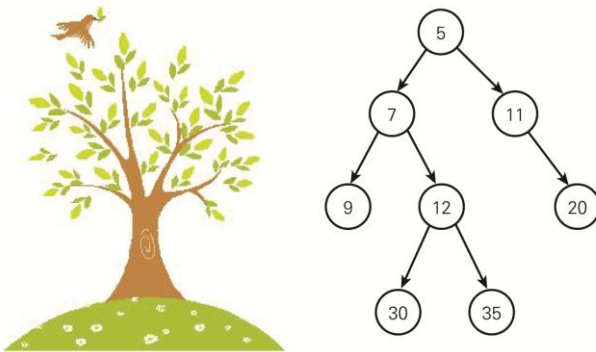
이전 회에서 설명했던 것은 가급적 또 설명하지 않을 테니 앞의 글들을 보지 않은 분들은 꼭 보기 바란다. 그럼 map에 대한 이야기를 시작한다.

### 8.1 map의 자료구조

map의 자료구조는 '트리<sup>tree</sup>'다(정확하게 말하면 트리 자료구조 중의 하나인 '레드-블랙 트리<sup>Red-Black tree</sup>'이다).

트리는 한글로 '나무'다. 나무는 뿌리에서 시작하여 여러 갈래의 가지가 있고, 가지의 끝에는 나무 잎이 있다. 트리 자료구조도 이와 같은 형태를 가지고 있어서 루트<sup>root</sup>, 리프<sup>leaf</sup>라는 용어를 사용한다.

그림 8-1 실제 나무(왼쪽)와 트리 자료구조의 모습(오른쪽)



오른쪽의 트리 자료구조에서 제일 최상 위의 '5'는 루트 노드<sup>root node</sup>라고 하며, 노드'5'와 노드'7'의 관계에서 노드'5'는 부모 노드<sup>parent node</sup>, 노드'7'은 자식 노드<sup>child node</sup>라고 한다. 또한 노드 '12'와 노드 '30'의 관계에서는 부모 노드는 노드'12'이다. 자식이 없는 노드는 리프 노드<sup>leaf node</sup>라고 한다. 그림 8-1에서는

'9', '30', '35', '20'이 리프 노드다.

## 8.2 트리 자료구조의 특징

트리는 노드를 균형 있게 가지는 것이 성능에 유리하기 때문에 기본 트리에서 변형된 B-트리, B+ 트리, R-트리, 레드 블랙 트리, AVL 트리 등 다양한 종류의 트리 자료구조가 있다.

균형을 이룬 트리는 자료를 정해진 방식에 따라서 분류하여 저장하기 때문에 시퀀스(일렬로)하게 자료를 저장하는 연결 리스트에 비해서 검색이 빠르다. 그렇지만 정해진 규칙에 따라서 자료를 삽입, 삭제 해야 되기 때문에 삽입과 삭제가 간단하지 않으며 구현이 복잡하다.

## 8.3 map을 언제 사용해야 될까?

map은 많은 자료를 정렬하여 저장하고 있고 빠른 검색을 필요로 할 때 자주 사용한다. 많은 자료를 빠르게 검색한다고 하는 부분은 앞 회에서 설명한 hash\_map과 비슷하다. 그러나 hash\_map과 크게 다른 부분이 있다. map은 자료를 저장할 때 내부에서 자동으로 정렬을 하고, hash\_map은 정렬하지 않는다는 것이다. 정렬이 필요하지 않는 곳에서 map을 사용하는 것은 불필요한 낭비이다.

map은 다음 조건일 때 사용하면 좋다.

1. 정렬해야 한다.
2. 많은 자료를 저장하고, 검색이 빨라야 한다
3. 빈번하게 삽입, 삭제하지 않는다.

## 8.4 map 사용방법

가장 먼저 map의 헤더파일을 포함한다.

```
#include <map>
```

보통 map을 사용하는 방법은 아래와 같다.

```
map< key 자료형, value 자료형 > 변수 이름  
map< int, int > map1;
```

value는 저장할 자료이고, key는 value를 가리키는 것이다. 위에서는 key의 자료형 int, value 자료형 int인 map을 생성한다.

앞에서 map은 자료를 저장할 때 정렬을 한다고 말했다. 정렬의 대상은 key를 대상으로 하며 오름차순으로 정렬한다. 그래서 내림차순으로 정렬하고 싶거나 key의 자료형이 기본형이 아닌 유저 정의형(class나 struct로 정의한 것)인 경우는 정렬 방법을 제공해야 한다.

위에 생성한 map1은 오름차순으로 정렬하는데 이것을 내림차순으로 정렬하고 싶다면 아래와 같이 하면 된다.

```
map< key 자료형, value 자료형, 비교 함수 > 변수 이름  
map< int, int, greater<int> > map1;
```

위에서 사용한 비교 함수 greater는 STL에 이미 정의되어 있는 템플릿이다.

greater와 같은 것을 STL 알고리즘 이라고 하는데 이것들은 10장에서 자세하게 설명할 예정이다. 여기서는 이런 것이 있다는 것만 알면 된다.

다른 컨테이너와 같이 map도 동적 할당을 할 수 있다. 사용방법은 앞서 소개한 컨테이너들과 비슷하다.

앞 회의 hash\_map과 비교를 하면 사용방법이 거의 같다는 것을 알 수 있다. 이후 소개하는 map의 멤버함수도 일부분만 제외하고 hash\_map과 같다. 이전 예도 이야기 했지만 서로 다른 컨테이너의 사용방법이 서로 비슷하여 하나만 제대로 배우면 나머지 것들도 배우기 쉽다는 것이 STL의 장점 중의 하나다.

## 8.4.1 map의 주요 멤버들

그림 8-2 map의 주요 멤버들

멤버	설명
begin	첫 번째 원소의 랜덤 접근 반복자를 리턴
clear	저장하고 있는 모든 원소를 삭제
empty	저장 하고 있는 요소가 없으면 true 리턴
End	마지막 원소 다음의(사용하지 않은 영역) 반복자를 리턴
erase	특정 위치의 원소나 지정 범위의 원소들을 삭제
Find	key 와 연관된 원소의 반복자 리턴
insert	원소 추가
lower_bound	지정한 key 의 요소를 가지고 있다면 해당 위치의 반복자를 리턴
operator[]	지정한 key 값으로 원소 추가 및 접근
rbegin	역방향으로 첫 번째 원소의 반복자를 리턴
rend	역방향으로 마지막 원소 다음의 반복자를 리턴
size	원소의 개수를 리턴
upper_bound	지정한 key 요소를 가지고 있다면 해당 위치 다음 위치의 반복자 리턴

## 8.4.2 추가

map에서는 자료를 추가 할 때 insert를 사용한다.

원형 :

```
pair <iterator, bool> insert( const value_type& _Val );  
iterator insert( iterator _Where, const value_type& _Val );  
template<class InputIterator> void insert( InputIterator _First, InputIterator  
_Last );
```

첫 번째 방식이 보통 가장 자주 사용하는 방식이다.

```
map< int, int > map1;
```

---

```
// key는 1, value는 35를 추가.  
map1.insert( map< int, int >::value_type(1, 35));
```

```
// 또는 STL의 pair를 사용하기도 한다.  
typedef pair< int, int > Int_Pair;  
map1.insert( Int_Pair(2, 45) );
```

두 번째 방식으로는 특정 위치에 추가할 수 있다.  
// 첫 번째 위치에 key 1, value 35를 추가  
map1.insert( map1.begin(), map< int, int >::value\_type(1, 35) );

```
// 또는  
map1.insert( map1.begin(), Int_Pair(2, 45) );
```

---

세 번째 방식으로는 지정한 반복자 구간에 있는 것들을 추가한다.

---

```
map< int, int > map2;  
// map1의 모든 요소를 map2에 추가.  
map2.insert( map1.begin(), map1.end() );
```

---

map은 이미 있는 key 값을 추가할 수 없다(복수의 key 값을 사용하기 위해서는 multi\_map을 사용해야 한다). 가장 자주 사용하는 첫 번째 방식으로 추가하는 경우는 아래와 같은 방법으로 결과를 알 수 있다.

---

```
pair< map<int, int>::iterator, bool > Result;  
Result = map1.insert( Int_Pair(1, 35));
```

---

만약 이미 key 값 1이 추가 되어 있었다면 insert 실패로 Result.second 는 false이며, 반대로 성공하였다면 true이다.

operator[]를 사용하여 추가하기

insert가 아닌 operator[]를 사용하여 추가할 수도 있다.

---

```
// key 10, value 80을 추가  
map1[10] = 80;
```

---

### 8.4.3 반복자 사용

다른 컨테이너와 같이 정 방향 반복자 `begin()`, `end()`와 역 방향 반복자 `rbegin()`, `rend()`를 지원한다. 사용방법은 다음과 같다.

---

```
// 정 방향으로 map1의 모든 요소의 value 출력
map< int, int >::iterator Iter_Pos;
for( Iter_Pos = map1.begin(); Iter_Pos != map1.end(); ++Iter_Pos)
{
    cout << Iter_Pos.second << endl;
}

// 역 방향으로 map1의 모든 요소의 value 출력
map< int, int >::reverse_iterator Iter_rPos;
for( Iter_rPos = map1.rbegin(); Iter_rPos != map1.rend(); ++Iter_rPos)
{
    cout << Iter_rPos.second << endl;
}
```

---

위에서 `map`을 정의할 때 비교함수를 사용할 수 있다고 했다. 만약 비교함수를 사용한 경우는 반복자를 정의할 때도 같은 비교함수를 사용해야 한다.

---

```
map< int, int, greater<int> > map1;
map< int, int, greater<int> >::iterator Iter_Pos;
```

---

### 8.4.4 검색

`map`에서 검색은 `key` 값을 대상으로 한다. `key`와 같은 요소를 찾으면 그 요소의 반복자를 리턴하고, 찾지 못한 경우에는 `end()`를 가리키는 반복자를 리턴한다.

---

원형 :

```
iterator find( const Key& _Key );
const_iterator find( const Key& _Key ) const;
```

---

두 방식의 차이는 리턴된 반복자가 `const`냐 아니냐의 차이이다. 첫 번째 방식은

const가 아니므로 찾은 요소의 value를 변경할 수 있다(참고로 절대 key는 변경 할 수 없다). 그러나 두 번째 방식은 value를 변경할 수 없다.

---

```
// key가 10인 요소 찾기.
map< int, int >::Iterator FindIter = map1.find( 10 );

// 찾았다면 value를 1000으로 변경
if( FindIter != map1.end() )
{
    FindIter->second = 1000;
}
```

---

## 8.4.5 삭제

저장하고 있는 요소를 삭제할 때는 erase와 clear를 사용한다. erase는 특정 요소를 삭제할 때 사용하고, clear는 모든 요소를 삭제할 때 사용한다.

### erase

---

원형 :

```
iterator erase( iterator _Where );
iterator erase( iterator _First, iterator _Last );
size_type erase( const key_type& _Key );
```

---

첫 번째 방식은 특정 위치에 있는 요소를 삭제한다.

---

```
// 두 번째 위치의 요소 삭제.
map1.erase( ++map1.begin() );
```

---

두 번째 방식은 지정한 구역에 있는 요소들을 삭제한다.

---

```
// map1의 처음과 마지막에 있는 모든 요소 삭제
map1.erase( map1.begin(), map1.end() );
```

---

세 번째 방식은 지정한 키와 같은 요소를 삭제한다.



---

```
// key가 10인 요소 삭제.  
map1.erase( 10 );
```

---

첫 번째와 두 번째 방식에서는 삭제하는 요소의 다음을 가리키는 반복자를 리턴하고(C++ 표준에서는 리턴하지 않지만 Microsoft Visual C++에서는 리턴한다), 세 번째 방식은 삭제된 개수를 리턴한다. map에서는 세 번째 방식으로 삭제를 하는 경우 정말 삭제가 되었다면 무조건 1이지만, multi\_map에서는 삭제한 개수만큼의 숫자가 나온다.

## clear

map의 모든 요소를 삭제할 때는 clear를 사용한다.

```
map1.clear();
```

이것으로 map에서 자주 사용하는 멤버들에 대한 설명은 끝났다. [표 8-1]에 나와 있는 멤버들 중 사용방법이 간단한 것은 따로 설명하지 않으니 [리스트 8-1]의 코드를 참고한다.

## [리스트 8-1] 정렬된 아이템 리스트 출력

---

```
#include <map>  
#include <string>  
#include <iostream>  
  
using namespace std;  
  
struct Item  
{  
    char Name[32]; // 이름  
    char Kind; // 종류  
    int BuyMoney; // 구입 가격  
    int SkillCd; // 스킬 코드  
};  
  
int main()  
{
```

---

---

```

map< char*, Item > Items;
map< char*, Item >::iterator IterPos;
typedef pair< char*, Item > ItemPair;

Item Item1;
strncpy( Item1.Name, "긴칼", 32 );
Item1.Kind = 1;   Item1.BuyMoney = 200;   Item1.SkillCd = 0;

Item Item2;
strncpy( Item2.Name, "성스러운 방패", 32 );
Item2.Kind = 2;   Item2.BuyMoney = 1000;   Item2.SkillCd = 4;

Item Item3;
strncpy( Item3.Name, "해머", 32 );
Item3.Kind = 1;   Item3.BuyMoney = 500;   Item3.SkillCd = 0;

// Items에 아이템 추가
Items.insert( map< char*, Item >::value_type(Item2.Name, Item2) );
Items.insert( ItemPair(Item1.Name, Item1) );

// Items가 비어 있지않다면
if( false == Items.empty() )
{
    cout << "저장된 아이템 개수- " << Items.size() << endl;
}

for( IterPos = Items.begin(); IterPos != Items.end(); ++IterPos )
{
    cout << "이름: " << IterPos->first << ", 가격: " << IterPos->second.BuyMoney << endl;
}

IterPos = Items.find("긴칼");
if( IterPos == Items.end() ) {
    cout << "아이템 '긴칼'이 없습니다." << endl;
}
cout << endl;

cout << "올림차순으로 정렬되어있는 map(Key 자료형으로string 사용)" << endl;

```

---

---

```

map< string, Item, less<string> > Items2;
map< string, Item, less<string> >::iterator IterPos2;

Items2.insert( map< string, Item >::value_type(Item2.Name, Item2) );
Items2.insert( ItemPair(Item1.Name, Item1) );
// operator[]를 사용하여 저장
Items2[Item3.Name] = Item3;

for( IterPos2 = Items2.begin(); IterPos2 != Items2.end(); ++IterPos2 )
{
    cout << "이름: " << IterPos2->first << ", 가격: " << IterPos2->second.BuyMoney << endl;
}
cout << endl;

cout << "해머의 가격은 얼마? ";
IterPos2 = Items2.find("해머");
if( IterPos2 != Items2.end() ) {
    cout << IterPos2->second.BuyMoney << endl;
}
else {
    cout << "해머는 없습니다" << endl;
}
cout << endl;

// 아이템 "긴칼"을 삭제한다.
IterPos2 = Items2.find("긴칼");
if( IterPos2 != Items2.end() ) {
    Items2.erase( IterPos2 );
}

cout << "Items2에 있는 아이템 개수: " << Items2.size() << endl;

return 0;
}

```

---

[리스트 8-1]의 실행 결과는 다음과 같다.

```
저장된 아이템 개수: 2
이름: 성스러운 방패, 가격: 1000
이름: 긴칼, 가격: 200
아이템 '긴칼' 이 없습니다.

올림차순으로 정렬되어있는 map<Key 자료형으로 string 사용>
이름: 긴칼, 가격: 200
이름: 성스러운 방패, 가격: 1000
이름: 해머, 가격: 500

해머의 가격은 얼마? 500

Items2에 있는 아이템 개수: 2
```

[리스트 8-1]의 Items에서 '긴칼'을 검색을 하면 찾을 수가 없다. 왜냐하면 key의 자료형으로 char\*을 사용했기 때문이다. 그래서 Items2에서는 STL의 문자열 라이브러리인 string을 사용했다. string에 대해서는 다음 기회에 설명할 예정이니 현재는 문자열을 처리하는 라이브러리라고 알고 있으면 된다. 이것으로 map에 대한 설명이 끝났다. 7장의 hash\_map과 비슷한 부분이 많아서 hash\_map에 대한 내용을 봤다면 쉽게 따라왔으리라 생각한다. 그리고 map과 hash\_map에 대하여 잘못 알고 있어서 정렬이 필요하지 않은 곳에서 map을 사용하는 경우가 있는데 조심하기 바란다. 여기에서 미처 설명하지 않은 부분은 마이크로소프트의 MSDN 사이트에 있는 map 설명을 참고한다.

MSDN 사이트: <http://msdn.microsoft.com/ko-kr/library/xdaye4c.aspx>

## 8.5 과제

1. [리스트 8-1]은 아이템 이름을 key 값으로 사용했다. 이번에는 아이템 가격을 Key 값으로 사용하여 아이템을 저장하고 내림차순으로 출력해 보라.

앞서 중복된 key를 사용할 수 있는 multi\_map 이라는 것이 있다고 했다. 이번 과제는 multi\_map을 사용해야 한다. multi\_map에 관한 설명은 아래 링크의 글을 참고한다.

<http://msdn.microsoft.com/ko-kr/library/1y9w8dz4.aspx>

## 9 셋(set)

앞서 8장에서 설명했던 map과 비슷하면서도 다른 set을 이번에 이야기 하려고 한다. map 이전에는 7장에서 hash\_map을 설명했다. 그런데 이번에 이야기할 set과 여러 부분에서 중복되는 부분이 있고, 저는 현업에서 set을 사용하는 경우가 거의 없어서 내용이 길지 않다.

### 9.1 set 이란

set은 원하는 key를 신속하게 찾고, 또 이 key가 정렬되기를 원할 때 사용한다 (여기서 말하는 key라는 것은 저장할 자료를 말한다). map과 비슷하지만 다른 점은 map은 key와 값을 한 쌍으로 저장하지만 set은 key만 저장한다. set도 map과 같이 key를 중복으로 저장할 수 없다. 만약 key를 중복으로 사용하고 싶다면 multiset을 사용해야 한다. 사용방법은 set과 거의 같다. set은 map과 같이 이진 탐색 트리 자료구조를 사용한다.

### 9.2 set을 사용할 때

앞서 이야기했듯이 set은 자료를 저장할 때 내부에서 자동으로 정렬하고, map과 다르게 key만 저장한다.

set은 다음과 같은 조건일 때 사용하면 좋다.

- 1 정렬해야 할 때
- 2 key가 있는지 없는지 알아야 할 때
- 3 많은 자료를 저장하고, 검색 속도가 빨라야 할 때

다른 컨테이너를 설명할 때 꽤 많은 이야기를 했는데 그동안 했던 이야기와 중복되는 것이 많고, 특히 앞 장의 map과 비슷한 부분이 많아서 이번에는 바로 사용방법으로 들어간다.

## 9.3 set 사용방법

set 컨테이너를 쓰려면 먼저 헤더 파일을 포함해야 한다.

```
#include <set>
```

보통 set을 사용하는 방법은 다음과 같다.

```
set< key 자료형> 변수 이름  
set< int> set1;
```

map과 사용방법이 비슷하죠? 다만, set은 위와 같이 key만 저장한다. 위에서는 key로 int 타입을 사용했다.

set은 map과 같이 기본적으로 오름차순으로 정렬을 한다. 만약 이것을 내림차순으로 바꾸고 싶거나 key의 자료형이 기본형이 아니란 사용자 정의형이라면 함수 객체로 정렬 방법을 제공해야 한다.

먼저 set의 key가 기본형이고 내림차순으로 정렬하고 싶다면 STL의 greater 알고리즘을 사용하면 된다.

```
set< key 자료형, 비교 함수> 변수 이름  
set< int, greater<int>> set1;
```

만약 key가 기본형이 아니고 Player 이라는 클래스를 사용하고 Player의 멤버 중 HP를 비교하여 정렬하고 싶다면 아래와 같이 하면 된다.

---

```
class Player  
{  
public:  
    Player() {}  
    ~Player() {}  
  
    int m_HP;  
};
```

```
template< typename T>
```

---

---

```

struct HP_COMPARE : public binary_function< T, T, bool >
{
    bool operator() (T& player1, T& player2) const
    {
        return player1.m_HP > player2.m_HP;
    }
};

int main()
{
    set< Player, HP_COMPARE<Player> > set1;
    return 0;
}

```

---

### 9.3.1 set의 주요 멤버들

표 9-1 set의 주요 멤버들

멤버	설명
begin	첫 번째 원소의 랜덤 접근 반복자를 리턴
clear	저장하고 있는 모든 원소를 삭제
empty	저장하고 있는 요소가 없으면 true 리턴
end	마지막 원소 다음의 (미 사용 영역) 반복자를 리턴
erase	특정 위치의 원소나 지정 범위의 원소들을 삭제
find	key 와 연관된 원소의 반복자 리턴
insert	원소 추가
lower_bound	지정한 key 의 요소를 가지고 있다면 해당 위치의 반복자를 리턴
operator[]	지정한 key 값으로 원소 추가 및 접근
rbegin	역방향으로 첫 번째 원소의 반복자를 리턴
rend	역방향으로 마지막 원소 다음의 반복자를 리턴
size	원소의 개수를 리턴
upper_bound	지정한 key 요소를 가지고 있다면 해당 위치 다음 위치의 반복자 리턴

## 9.3.2 추가

set 에서는 자료를 추가 할 때 insert를 사용한다.

---

원형 :

```
pair<iterator, bool> insert( const value_type& _Val );  
iterator insert( iterator _Where, const value_type& _Val );  
template<class InputIterator> void insert( InputIterator _First, InputIterator _Last );
```

---

첫 번째가 자주 사용하는 방식이다.

---

```
set< int > set1;  
  
// key 1을 추가.  
set1.insert( 1 );  
  
// 추가했는지 조사 하고 싶을 때는  
pair< set<int>::iterator, bool > Result;  
Result = set1.insert( 1 );  
if( Result.second )  
{  
    // 추가 성공  
}  
else  
{  
    // 추가 실패  
}
```

---

두 번째 방식은 특정 위치에 추가할 수 있다.

---

```
// 첫 번째 위치에 key 1, value 10를 추가  
set1.insert( set1.begin(), 10 );
```

---

세 번째 방식은 지정한 반복자 구간에 있는 것들을 추가한다.

---

```
set< int > set2;
```

---



---

```
// set1의 모든 요소를 set2에 추가.  
set2.insert( set1.begin(), set1.end() );
```

---

set은 이미 있는 key 값을 추가할 수 없다(복수의 key 값을 사용하기 위해서는 multiset을 사용해야 한다). 참고로 특정 위치를 지정하여 추가를 하여도 정렬되어 저장한다.

#### [리스트 9-1] 특정 위치에 추가했을 때의 정렬 여부

---

```
#include <iostream>  
#include <functional>  
#include <set>  
  
using namespace std;  
  
int main()  
{  
    set< int > set1;  
    set1.insert( 10 );  
    set1.insert( 15 );  
    set1.insert( 12 );  
    set1.insert( 2 );  
    set1.insert( 100 );  
  
    for( set<int>::iterator IterPos = set1.begin();  
        IterPos != set1.end(); ++IterPos )  
    {  
        cout << *IterPos << endl;  
    }  
  
    set<int>::iterator IterPos = set1.begin();  
    ++IterPos;  
    set1.insert( IterPos, 90 );  
  
    cout << endl;  
    cout << "90을 추가 후 set1의 모든 요소 출력" << endl;  
    for( set<int>::iterator IterPos = set1.begin();  
        IterPos != set1.end(); ++IterPos )  
    {
```

---

---

```

        cout << *IterPos << endl;
    }

    return 0;
}

```

---

[리스트 9-1]의 실행 결과는 다음과 같다.

```

2
10
12
15
100

90을 추가 후 set1의 모든 요소 출력
2
10
12
15
90
100

```

### 9.3.3 반복자 사용

다른 컨테이너와 같이 정 방향 반복자를 사용하는 `begin()`, `end()`와 역 방향 반복자를 사용하는 `rbegin()`, `rend()`를 지원한다. 사용방법은 다음과 같다.

---

```

// 정 방향으로 set1의 모든 Key 출력
set< int >::iterator Iter_Pos;
for( Iter_Pos = set1.begin(); Iter_Pos != set1.end(); ++Iter_Pos)
{
    cout << *Iter_Pos << endl;
}

// 역 방향으로 set1의 모든 요소 Key 출력
set< int >::reverse_iterator Iter_rPos;
for( Iter_rPos = set1.rbegin(); Iter_rPos != set1.rend(); ++Iter_rPos)
{
    cout << *Iter_rPos << endl;
}

```

---

### 9.3.4 검색

set에서 검색은 key를 대상으로 한다. key와 같은 요소를 찾으면 그 요소의 반복자를 리턴하고, 찾지 못한 경우에는 end()를 가리키는 반복자를 리턴한다.

---

원형 :

```
iterator find( const Key& _Key );  
const_iterator find( const Key& _Key ) const;
```

---

두 방식의 차이는 리턴된 반복자가 const 여부다. 첫 번째 방식은 const가 아니므로 찾은 요소의 key를 변경할 수 있다. 그러나 두 번째 방식은 key를 변경할 수 없다.

---

```
// key가 10인 요소 찾기.  
set< int >::Iterator FindIter = set1.find( 10 );  
  
// 찾았다면 value를 변경  
if( FindIter != set1.end() )  
{  
    // Key를 찾았다!!!  
}
```

---

set은 map과 다르게 Key만 저장하기에 Key의 변경이 가능하지만 find로 찾은 Key를 변경하면 정렬되지 않는다.

[리스트 9-2] find로 찾은 Key 변경

---

```
#include <iostream>  
#include <functional>  
#include <set>  
  
using namespace std;  
  
int main()  
{  
    set< int > set1;  
    set1.insert( 10 );
```

---

---

```

set1.insert( 15 );
set1.insert( 12 );

for( set<int>::iterator IterPos = set1.begin();
    IterPos != set1.end(); ++IterPos )
{
    cout << *IterPos << endl;
}

set<int>::iterator FindIter = set1.find( 15 );
if( FindIter != set1.end() )
{
    *FindIter = 11;
}

cout << endl;
cout << "15를 검색 후 11로 변경한 후 set1의 모든 요소 출력" << endl;
for( set<int>::iterator IterPos = set1.begin();
    IterPos != set1.end(); ++IterPos )
{
    cout << *IterPos << endl;
}

return 0;
}

```

---

[리스트 9-2]의 실행 결과는 다음과 같다.

```

10
12
15

15를 검색 후 11로 변경한 후 set1의 모든 요소 출력
10
12
11

```

### 9.3.5 삭제

저장하고 있는 요소를 삭제할 때는 erase와 clear를 사용한다. erase는 특정 요소를 삭제할 때 사용하고, clear는 모든 요소를 삭제할 때 사용한다.

## erase

원형 :

```
iterator erase( iterator _Where );  
iterator erase( iterator _First, iterator _Last );  
size_type erase( const key_type& _Key );
```

첫 번째 방식은 특정 위치에 있는 요소를 삭제한다.

```
// 두 번째 위치의 요소 삭제  
set1.erase( ++set1.begin() );
```

두 번째 방식은 지정한 구간에 있는 요소들을 삭제한다.

```
// set1의 처음과 마지막에 있는 모든 요소 삭제  
set1.erase( set1.begin(), set1.end() );
```

세 번째 방식은 지정한 Key와 같은 요소를 삭제한다.

```
// key가 10인 요소 삭제  
set1.erase( 10 );
```

첫 번째와 두 번째 방식으로 삭제를 하면 삭제되는 요소의 다음을 가리키는 반복자를 리턴한다. 세 번째 방식은 삭제된 개수를 리턴하는 데 정말 삭제가 되었다면 1이 리턴된다. 그러나 multiset에서는 1이 아닌 삭제된 개수를 리턴한다.

## clear

set의 모든 요소를 삭제할 때는 다음과 같이 clear를 사용한다.

```
set1.clear();
```

이것으로 set에서 자주 사용하는 멤버들에 대한 설명은 끝났다. 아래의 [리스트 9-3]은 set을 전반적으로 사용하는 예를 나타내고 있다.

#### [리스트 9-3] set 사용 예

---

```
#include <iostream>
#include <functional>
#include <set>

using namespace std;

class Player
{
public:
    Player() {}
    ~Player() {}

    int m_Level;
};

// 레벨이 높은 순으로 정렬
template< typename T >
struct LEVEL_COMPARE : public binary_function< T, T, bool >
{
    bool operator() (const T& player1, const T& player2) const
    {
        return player1->m_Level > player2->m_Level;
    }
};

int main()
{
    set< Player*, LEVEL_COMPARE<Player*> > PlayerList;

    Player* pPlayer1 = new Player; pPlayer1->m_Level = 10;
    PlayerList.insert( pPlayer1 );
    Player* pPlayer2 = new Player; pPlayer2->m_Level = 45;
    PlayerList.insert( pPlayer2 );
    Player* pPlayer3 = new Player; pPlayer3->m_Level = 5;
```

---

---

```

PlayerList.insert( pPlayer3 );
Player* pPlayer4 = new Player; pPlayer4->m_Level = 15;
PlayerList.insert( pPlayer4 );

// 정 방향으로 출력( 레벨이 높은 순으로)
for( set< Player*, LEVEL_COMPARE<Player*> >::iterator IterPos =
PlayerList.begin();
    IterPos != PlayerList.end(); ++IterPos )
{
    cout << (*IterPos)->m_Level << endl;
}

cout << endl;

// 역 방향으로 출력( 레벨이 낮은 순으로)
for( set< Player*, LEVEL_COMPARE<Player*> >::reverse_iterator IterPos = PlayerList.rbegin();
    IterPos != PlayerList.rend(); ++IterPos )
{
    cout << (*IterPos)->m_Level << endl;
}

cout << endl;

// pPlayer4를 검색
set< Player*, LEVEL_COMPARE<Player*> >::iterator FindPlayer = PlayerList.find( pPlayer4 );
if( FindPlayer != PlayerList.end() )
{
    cout << "pPlayer4를 찾았습니다" << endl;

    cout << "pPlayer4 삭제" << endl;
    PlayerList.erase( FindPlayer );
}
else
{
    cout << "pPlayer4를 못찾았습니다" << endl;
}

cout << endl;
cout << "Total Player Count : " << PlayerList.size() << endl;

```

---

---

```

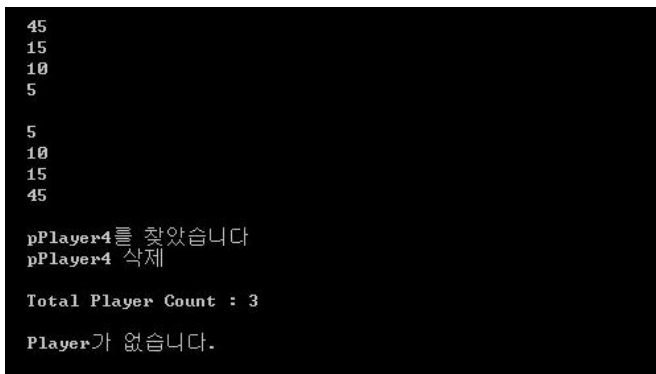
    cout << endl;
    PlayerList.clear();
    if( PlayerList.empty() )
    {
        cout << "Player가 없습니다." << endl;
    }

    delete pPlayer1;
    delete pPlayer2;
    delete pPlayer3;
    delete pPlayer4;
    return 0;
}

```

---

[리스트 9-3]의 실행 결과는 다음과 같다.



```

45
15
10
5

5
10
15
45

pPlayer4를 찾았습니다
pPlayer4 삭제

Total Player Count : 3

Player가 없습니다.

```

앞 장의 map과 거의 대부분 비슷하기 때문에 map을 아시는 분들은 아주 쉬웠을 것이라고 생각한다.

앞 장에서와 마찬가지로 set의 멤버 중 설명하지 않은 것들은 MSDN에 있는 set 설명을 참조하시기를 바란다.



## 9.4 과제

set은 key를 중복으로 저장할 수 없습니다. 중복 Key를 저장하기 위해서는 multiset을 사용해야 한다. multiset을 공부해 보세요.

참고: [http://msdn.microsoft.com/ko-kr/library/vstudio/5cktszy1\(v=vs.100\).aspx](http://msdn.microsoft.com/ko-kr/library/vstudio/5cktszy1(v=vs.100).aspx)

## 10 알고리즘

template부터 시작하여 이전 장까지는 STL의 컨테이너들에 대해 설명했다. 이제 알고리즘에 대한 것만 남았다. 필자가 쓴 글이 C++ STL을 공부하는데 조금 이나마 도움이 되기를 바란다.

이 장에서는 STL의 알고리즘 중 많이 사용하고 있는 것을 중심으로 설명한다. STL의 컨테이너는 그 자체만으로도 대단히 유용한 것이지만 알고리즘과 결합되면 유용성은 더욱 더 커진다.

STL의 알고리즘은 컨테이너처럼 제네릭(총칭적)한 특징이 있는데 그 이유는 STL에 있는 다양한 알고리즘을 한 종류의 컨테이너에만 사용할 수 있는 것이 아닌 vector, list, deque, map 등 여러 가지 다양한 컨테이너에 사용할 수 있기 때문이다(참고로 STL 컨테이너만이 아닌 일반 배열 구조도 알고리즘에 사용할 수 있다).

### 10.1 STL 알고리즘 분류

STL 알고리즘은 크게 4가지로 분류할 수 있다.

- 변경 불가 시퀀스 알고리즘
- 변경 가능 시퀀스 알고리즘
- 정렬 관련 알고리즘
- 범용 수치 알고리즘

먼저, 변경 불가 시퀀스 알고리즘에는 find와 for\_each 등이 있으며 대상 컨테이너의 내용을 변경하지 못한다. 그리고 변경 가능 시퀀스 알고리즘으로는 copy, generate 등이 있으며 대상 컨테이너 내용을 변경한다. 또한, 정렬 관련 알고리즘은 정렬이나 머지를 하는 알고리즘으로 sort와 merge 등이 있다. 끝으로, 범용 수치 알고리즘은 값을 계산하는 알고리즘으로 accumulate 등이 있다.

STL의 모든 알고리즘을 설명하기에는 많은 분량이 필요하므로 위에 소개한 알고리즘 카테고리 별로 그 중에서 자주 사용하는 알고리즘 중심으로 어떤 기능이 있으며 어떻게 사용하는지 설명하도록 한다.

## 10.2 조건자

알고리즘 중에는 함수를 파라미터로 받아들이는 것이 많다. 알고리즘에서 함수를 파라미터로 받아들이지 않거나 기본 함수 인자를 사용하며 알고리즘을 사용하는 컨테이너의 데이터는 기본 자료 형만을 사용할 수 있다. 사용자 정의형 자료 형(struct, class)을 담는 컨테이너를 알고리즘에서 사용하기 위해서는 관련 함수를 만들어서 파라미터로 넘겨줘야 한다. 알고리즘에 파라미터로 넘기는 함수는 보통 함수보다는 함수 객체를 사용하는 편이다.

## 10.3 변경 불가 시퀀스 알고리즘

### 10.3.1 find

컨테이너에 있는 데이터 중 원하는 것을 찾기 위해서는 find 알고리즘을 사용하면 된다. 참고로 알고리즘을 사용하려면 algorithm 헤더 파일을 추가해야 한다.

```
#include < algorithm >
```

#### find의 원형

---

```
template<class InputIterator, class Type>
    InputIterator find( InputIterator _First, InputIterator _Last, const Type&
        _Val );
```

---

첫 번째 파라미터에 찾기를 시작할 반복자 위치, 두 번째 파라미터에 마지막 위치(반복자의 end())와 같이 이 위치의 바로 앞까지 검색함), 세 번째 파라미터에는 찾을 값을 넘긴다.

find가 성공하면 데이터가 있는 위치를 가리키는 반복자를 반환하고, 실패하면 반복자의 end()를 반환한다.

#### find 사용방법

---

```
vector< int > ItemCodes;
.....
```

---

---

```
// ItemCodes 컨테이너의 시작과 끝 사이에서 15를 찾는다.  
find( ItemCodes.begin(), ItemCodes.end(), 15 );
```

---

find를 사용하여 캐릭터의 아이템을 검색하는 예제 코드다.

### [리스트 10-1] 캐릭터 아이템 검색

---

```
#include <algorithm>  
#include <vector>  
#include <iostream>  
  
using namespace std;  
  
int main()  
{  
    vector< int > CharItems;  
    CharItems.push_back( 12 );  
    CharItems.push_back( 100 );  
    CharItems.push_back( 77 );  
  
    vector< int >::iterator FindIter;  
  
    // CharItems의 처음과 끝에서 12를 찾는다.  
    FindIter = find( CharItems.begin(), CharItems.end(), 12 );  
    if( FindIter != CharItems.end() )  
    {  
        cout << "CharItem 12를 찾았습니다." << endl;  
    }  
    else  
    {  
        cout << "CharItem 12는 없습니다" << endl;  
    }  
  
    // CharItems 두 번째와 끝에서 12를 찾는다.  
    // ++CharItems.begin()로 반복자를 한칸 이동시킨다.  
    FindIter = find( ++CharItems.begin(), CharItems.end(), 12 );  
    if( FindIter != CharItems.end() )  
    {  
        cout << "CharItem 12를 찾았습니다." << endl;  
    }  
}
```

---

---


```

    }
    else
    {
        cout << "CharItem 12는 없습니다" << endl;
    }
    return 0;
}

```

---

[리스트 10-1]의 결과는 다음과 같다.



```

CharItem 12를 찾았습니다.
CharItem 12는 없습니다

```

[리스트 10-1]에서는 vector 컨테이너를 대상으로 했지만 vector 이외의 list, deque도 같은 방식으로 사용한다. 다만 map, set, hash\_map의 연관 컨테이너는 해당 컨테이너의 멤버로 find()를 가지고 있으므로 알고리즘에 있는 find가 아닌 해당 멤버 find를 사용한다.

### 10.3.2 find\_if

find를 사용하면 컨테이너에 있는 데이터 중 찾기 원하는 것을 쉽게 찾을 수 있다. 그런데 find만으로는 많이 부족하다. 이유는 find는 기본형만을 컨테이너에 저장하고 있을 때만 사용할 수 있다. 만약 유저 정의형을 저장하는 컨테이너는 find를 사용할 수 없다. 이럴 때는 10.2에서 짧게 설명한 조건자를 사용해야 한다. 즉, 조건자에 어떤 방식으로 찾을지 정의하고, 알고리즘에서 이 조건자를 사용한다. find\_if는 기본적으로 find와 같으며, 다른 점은 조건자를 받아들이는 것이다.

#### find\_if 원형

---

```

template<class InputIterator, class Predicate>
InputIterator find_if( InputIterator _First, InputIterator _Last, Predicate
_Pred );

```

---

find와 비교하면 첫 번째와 두 번째 파라미터는 동일하며, 세 번째 파라미터

가 다르다. find는 세 번째 파라미터에 찾기 원하는 값을 넘기지만 find\_if는 조건자를 넘긴다.

## find\_if 사용법

---

```
// 컨테이너에 저장할 유저 정의형
struct User
{
    .....,
    int Money;
    int MobKillCount;
    .....
};

// 조건자
struct FindBestUser()
{
    .....,
};

vector< User > Users;
.....

find_if( Users.begin(), Users.end(), FindBestUser() );
```

---

조건자가 있다는 것 이외에는 find\_if는 find와 같으므로 조건자 부분만 잘 보면 된다.

## [리스트 10-2] 특정 돈을 가지고 있는 유저 찾기

---

```
#include <algorithm>
#include <vector>
#include <iostream>

using namespace std;

struct User
{
```

---

---

```

    int Money;
    int Level;
};

struct FindMoneyUser
{
    bool operator() ( User& user ) const { return user.Money == CompareMoney; }
    int CompareMoney;
};

int main()
{
    vector< User > Users;

    User user1; user1.Level = 10; user1.Money = 2000;
    User user2; user2.Level = 5;  user2.Money = -10;
    User user3; user3.Level = 20; user3.Money = 35000;

    Users.push_back( user1 );
    Users.push_back( user2 );
    Users.push_back( user3 );

    vector< User >::iterator FindUser;

    FindMoneyUser tFindMoneyUser;
    tFindMoneyUser.CompareMoney = 2000;
    FindUser = find_if( Users.begin(), Users.end(), tFindMoneyUser );
    if( FindUser != Users.end() )
    {
        cout << "소지하고 있는 돈은: " << FindUser->Money << "입니다" << endl;
    }
    else
    {
        cout << " 유저가 없습니다. " << endl;
    }
    return 0;
}

```

---

[리스트 10-2]의 결과는 다음과 같다.

소지하고 있는 돈은: 2000입니다

[리스트 10-2]에서는 조건자 함수를 함수 포인터가 아닌 함수 객체를 사용하였다. 함수 객체를 사용한 이유는 함수 객체는 inline화 되므로 함수 포인터와 비교하면 함수 포인터 참조와 함수 호출에 따른 작업이 필요 없으며, 함수 객체를 사용하면 객체에 특정 데이터를 저장할 수 있어서 조건자가 유연해진다.

[리스트 10-2]의 FindMoneyUser를 일반 함수로 만들었다면 비교로 사용할 CompareMoney의 값은 함수를 정의할 때 고정된다. 그러나 함수 객체로 만들면 객체를 생성할 때 CompareMoney의 값을 설정할 수 있어서 유연해진다.

**NOTE** 조건자를 사용하는 알고리즘 : STL의 알고리즘은 조건자를 사용하지 않는 것과 조건자를 사용하는 알고리즘 두 가지 버전을 가지고 있는 알고리즘이 있다. 이중 조건자를 사용하는 알고리즘은 보통 조건자를 사용하지 않는 알고리즘의 이름에서 뒤에 '\_if'를 붙인 이름으로 되어 있다.

### 10.3.3 for\_each

for\_each는 순차적으로 컨테이너들에 담긴 데이터를 함수의 파라미터로 넘겨서 함수를 실행시키는 알고리즘이다.

온라인 게임에서 예를 들면 플레이하고 있는 유저 객체를 vector 컨테이너인 Users에 저장하고 모든 유저들의 현재 플레이 시간을 갱신하는 기능을 구현한다면 플레이 시간을 계산하는 함수 객체 UpdatePlayTime을 만든 후 for\_each에 Users의 시작과 끝 반복자와 UpdatePlayTime 함수 객체를 넘기는 방식이다.

#### for\_each 원형

```
template<class InputIterator, class Function>
Function for_each( InputIterator _First, InputIterator _Last, Function _Func );
```

첫 번째 파라미터는 함수의 파라미터로 넘길 시작 위치, 두 번째 파라미터는



함수의 인자로 넘길 마지막 위치, 세 번째는 컨테이너의 데이터를 파라미터로 받을 함수다.

## for\_each 사용방법

---

```
// 함수 객체
struct UpdatePlayTime
{
    .....
};

vector< User > Users;

for_each( Users.begin(), Users.end(), UpdatePlayTime );
```

---

다음 [리스트 10-3]은 위에서 예를 들어 설명한 부분을 완전하게 구현한 예제 코드다.

[리스트 10-3] for\_each를 사용하여 유저들의 플레이 시간 갱신

---

```
#include <algorithm>
#include <vector>
#include <iostream>

using namespace std;

struct User
{
    int UID;
    int PlayTime;
};

struct UpdatePlayTime
{
    void operator() ( User& user )
    {
        user.PlayTime += PlayTime;
    }
}
```

---

---

```

    int PlayTime;
};

int main()
{
    vector< User > Users;

    User user1; user1.UID = 1; user1.PlayTime = 40000;
    User user2; user2.UID = 2; user2.PlayTime = 0;
    User user3; user3.UID = 3; user3.PlayTime = 25000;

    Users.push_back( user1 );
    Users.push_back( user2 );
    Users.push_back( user3 );

    // 현재 플레이 시간
    vector< User >::iterator IterUser;
    for( IterUser = Users.begin(); IterUser != Users.end(); ++IterUser )
    {
        cout << "UID : " << cout << IterUser->UID << "의 총 플레이 시간: " <<
            IterUser->PlayTime << endl;
    }
    cout << endl;

    UpdatePlayTime updatePlayTime;
    updatePlayTime.PlayTime = 200;

    // 두 번째 유저부터 갱신
    for_each( Users.begin() + 1, Users.end(), updatePlayTime );

    for( IterUser = Users.begin(); IterUser != Users.end(); ++IterUser )
    {
        cout << "UID : " << cout << IterUser->UID << "의 총 플레이 시간: " <<
            IterUser->PlayTime << endl;
    }

    return 0;
}

```

---

[리스트 10-3]의 결과는 다음과 같다.

```
UID : 1052ED481의 총 레이 시간: 40000
UID : 1052ED482의 총 레이 시간: 0
UID : 1052ED483의 총 레이 시간: 25000

UID : 1052ED481의 총 레이 시간: 40000
UID : 1052ED482의 총 레이 시간: 200
UID : 1052ED483의 총 레이 시간: 25200
```

변경 불가 시퀀스 알고리즘에는 위에 설명한 find, find\_if, for\_each 이외에 count, search, equal, adjacent\_find, equal 등이 있다.

## 10.4 변경 가능 시퀀스 알고리즘

위 알고리즘들이 대상이 되는 컨테이너에 저장한 데이터를 변경하지 않는 것들이라면 이번에 설명한 알고리즘들은 제목대로 대상이 되는 컨테이너의 내용을 변경하는 알고리즘들이다. 컨테이너에 복사, 삭제, 대체 등을 할 때는 이번에 소개할 알고리즘들을 사용한다.

### 10.4.1 generate

컨테이너의 특정 구간을 특정 값으로 채우고 싶을 때가 있다. 이 값이 동일한 것이라면 컨테이너의 assign() 멤버를 사용하면 되지만 동일한 값이 아니라면 assign()을 사용할 수 없다.

이 때 사용하는 알고리즘이 generate이다. generate 알고리즘에 값을 채울 컨테이너의 시작과 끝, 값을 생성할 함수를 파라미터로 넘긴다.

#### generate 원형

---

```
template<class ForwardIterator, class Generator>
void generate( ForwardIterator _First, ForwardIterator _Last, Generator _Gen );
```

---

첫 번째 파라미터는 값을 채울 컨테이너의 시작 위치의 반복자, 두 번째는 컨테이너의 마지막 위치의 반복자, 세 번째는 값을 생성할 함수다.

## generate 사용방법

---

```
// 값 생성 함수
struct SetUserInfo
{
    .....
};

vector< User > Users(5);

generate( Users.begin(), Users.end(), SetUserInfo() );
```

---

generate 알고리즘의 대상이 되는 컨테이너는 값을 채울 공간이 미리 만들어져 있어야 한다. 즉 generate는 컨테이너에 데이터를 추가하는 것이 아니고 기존의 데이터를 다른 값으로 변경하는 것이다.

[리스트 10-4] generate를 사용하여 유저의 기초 데이터 설정

---

```
#include <algorithm>
#include <vector>
#include <iostream>

struct User
{
    int UID;
    int RaceType;
    int Sex;
    int Money;
};

struct SetUserInfo
{
    SetUserInfo() { UserCount = 0; }

    User operator() ()
    {
        User user;
```

---

---

```
++UserCount;

user.UID = UserCount;
user.Money = 2000;

if( 0 == (UserCount%2) )
{
    user.RaceType = 1;
    user.Sex = 1;
    user.Money += 1000;
}
else
{
    user.RaceType = 0;
    user.Sex = 0;
}

return user;
}

int UserCount;
};

int main()
{
    vector< User > Users(5);

    generate( Users.begin(), Users.end(), SetUserInfo() );

    char szUserInfo[256] = {0,};

    vector< User >::iterator IterUser;
    for( IterUser = Users.begin(); IterUser != Users.end(); ++IterUser )
    {
        sprintf( szUserInfo, "UID %d, RaceType : %d, Sex : %d, Money : %d",
            IterUser->UID, IterUser->RaceType, IterUser->Sex, IterUser->Money );

        cout << szUserInfo << endl;
    }
}
```

---

---

```
    return 0;
}
```

---

[리스트 10-4]의 결과는 다음과 같다.

```
UID 1, RaceType : 0, Sex : 0, Money : 2000
UID 2, RaceType : 1, Sex : 1, Money : 3000
UID 3, RaceType : 0, Sex : 0, Money : 2000
UID 4, RaceType : 1, Sex : 1, Money : 3000
UID 5, RaceType : 0, Sex : 0, Money : 2000
```

## 10.4.2 copy

copy 알고리즘은 컨테이너에 저장한 것과 같은 자료형을 저장하는 다른 컨테이너에 복사하고 싶을 때 사용한다.

컨테이너 A의 데이터를 컨테이너 B에 copy 하는 경우 컨테이너 B에 데이터를 추가하는 것이 아니고 덮쓰는 것이므로 A에서 10개를 복사하는 경우 B에는 10개만큼의 공간이 없다면 버그가 발생한다. 또 A와 B 컨테이너는 같은 컨테이너일 필요는 없지만 당연히 컨테이너에 저장하는 자료 형은 같아야 한다.

---

copy 원형

```
template<class InputIterator, class OutputIterator>
OutputIterator copy(
    InputIterator _First,
    InputIterator _Last,
    OutputIterator _DestBeg
);
```

---

첫 번째와 두 번째 파라미터는 복사하려는 입력 구간의 시작과 마지막을 가리키는 반복자다. 세 번째 파라미터는 붙여 넣을 출력 구간의 시작 위치를 가리키는 반복자다.

## copy 사용방법

---

```
vector<int> vec1;
.....
vector<int> vec2;

copy( vec1.begin(), vec1.end(), vec2.begin() );
```

---

### [리스트 10-5] copy 사용 예

---

```
#include <algorithm>
#include <vector>
#include <list>
#include <iostream>

using namespace std;

int main()
{
    vector<int> vec1(10);
    generate( vec1.begin(), vec1.end(), rand );

    cout << "vec1의 모든 데이터를 vec2에 copy" << endl;

    vector<int> vec2(10);
    copy( vec1.begin(), vec1.end(), vec2.begin() );
    for( vector<int>::iterator IterPos = vec2.begin();
        IterPos != vec2.end();
        ++IterPos )
    {
        cout << *IterPos << endl;
    }
    cout << endl;

    cout << "vec1의 모든 데이터를 list1에 copy" << endl;
    list<int> list1(10);
    copy( vec1.begin(), vec1.end(), list1.begin() );

    for( list<int>::iterator IterPos2 = list1.begin();
```

---

---

```

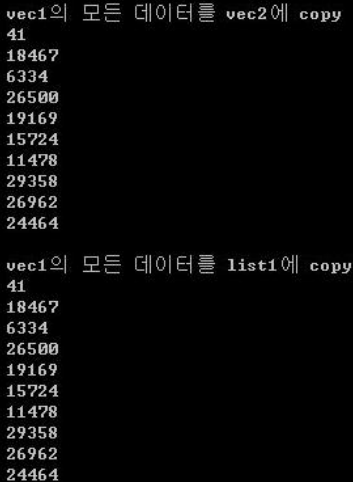
    IterPos2 != list1.end();
    ++IterPos2 )
{
    cout << *IterPos2 << endl;
}

return 0;
}

```

---

[리스트 10-5]의 실행 결과는 다음과 같다.



```

vec1의 모든 데이터를 vec2에 copy
41
18467
6334
26500
19169
15724
11478
29358
26962
24464

vec1의 모든 데이터를 list1에 copy
41
18467
6334
26500
19169
15724
11478
29358
26962
24464

```

### 10.4.3 remove

remove 알고리즘은 컨테이너에 있는 특정 값들을 삭제하고 싶은 때 사용한다. 주의해야 될 점은 삭제 후 크기가 변하지 않는다는 것이다. 삭제가 성공하면 삭제 대상이 아닌 데이터들을 앞으로 옮겨 놓고 마지막 위치의(컨테이너의 end())가 아닌 삭제 후 빈 공간에 다른 데이터를 쓰기 시작한 위치) 반복자를 반환한다. 리턴 값이 가리키는 부분부터 끝(end()) 사이의 데이터 순서는 정의되어 있지 않으며 진짜 삭제를 하기 위해서는 erase()를 사용해야 한다.



## remove 원형

---

```
template<class ForwardIterator, class Type>
ForwardIterator remove( ForwardIterator _First, ForwardIterator _Last, const
Type& _Val );
```

---

첫 번째 파라미터는 삭제 대상을 찾기 시작할 위치의 반복자, 두 번째 파라미터는 삭제 대상을 찾을 마지막 위치의 반복자, 세 번째 파라미터는 삭제할 값이다.

## remove 사용방법

---

```
vector<int> vec1;
...
remove( vec1.begin(), vec1.end(), 20 );
```

---

### [리스트 10-6] remove 사용 예

---

```
#include <algorithm>
#include <vector>
#include <list>
#include <iostream>

using namespace std;

int main()
{
    vector<int> vec1;
    vec1.push_back(10); vec1.push_back(20); vec1.push_back(20);
    vec1.push_back(40); vec1.push_back(50); vec1.push_back(30);

    vector<int>::iterator iterPos;

    cout << "vec1에 있는 모든 데이터 출력" << endl;
    for( iterPos = vec1.begin(); iterPos != vec1.end(); ++iterPos )
    {
        cout << *iterPos << " " << endl;
    }
}
```

---

---

```
}
cout << endl;

cout << "vec1에서 20 remove" << endl;
vector<int>::iterator RemovePos = remove( vec1.begin(), vec1.end(), 20 );

for( iterPos = vec1.begin(); iterPos != vec1.end(); ++iterPos )
{
    cout << *iterPos << " " << endl;
}
cout << endl;

cout << "vec1에서 20 remove 이후 사용 하지않는 영역 완전 제거" << endl;
while( RemovePos != vec1.end() )
{
    RemovePos = vec1.erase( RemovePos );
}

for( iterPos = vec1.begin(); iterPos != vec1.end(); ++iterPos )
{
    cout << *iterPos << " " << endl;
}
}
```

---

[리스트 10-6]의 실행 결과는 다음과 같다.

vec1에 있는 모든 데이터 출력

10  
20  
20  
40  
50  
30

vec1에서 20 remove

10  
40  
50  
30  
50  
30

vec1에서 20 remove 이후 사용 하지않는 영역 완전 제거

10  
40  
50  
30

[리스트 10-6]에서는 remove 후 erase로 완전하게 제거하는 예를 보여준다.

```
vector<int>::iterator RemovePos = remove( vec1.begin(), vec1.end(), 20 );
```

로 삭제 후 남은 영역의 반복자 위치를 받은 후 다음과 같이 완전하게 제거한다.

---

```
while( RemovePos != vec1.end() )  
{  
    RemovePos = vec1.erase( RemovePos );  
}
```

---

## 10.4.4 replace

컨테이너의 특정 값을 다른 값으로 바꾸고 싶을 때는 replace 알고리즘을 사용한다.

---

replace 원형

---

---

```
template<class ForwardIterator, class Type>
void replace(
    ForwardIterator _First,
    ForwardIterator _Last,
    const Type& _OldVal,
    const Type& _NewVal
);
```

---

첫 번째 파라미터는 교체 대상을 찾기 시작할 위치의 반복자, 두 번째 파라미터는 교체 대상을 찾을 마지막 위치의 반복자, 세 번째 파라미터는 교체하기 이전의 값, 네 번째 파라미터는 교체할 값이다.

## replace 사용방법

---

```
vector<int> vec1;
.....
replace( vec1.begin(), vec1.end(), 20, 30 );
```

---

## [리스트 10-7] replace 사용 예

---

```
#include <algorithm>
#include <vector>
#include <list>
#include <iostream>

using namespace std;

int main()
{
    vector<int> vec1;
    vec1.push_back(10); vec1.push_back(20); vec1.push_back(20);
    vec1.push_back(40); vec1.push_back(50); vec1.push_back(30);

    vector<int>::iterator iterPos;

    cout << "vec1에 있는 모든 데이터 출력" << endl;
    for( iterPos = vec1.begin(); iterPos != vec1.end(); ++iterPos )
    {
```

---

---

```

    cout << *iterPos << " " << endl;
}
cout << endl;

cout << "vec1의 세 번째 요소부터 20을 200으로 변경" << endl;
replace( vec1.begin() + 2, vec1.end(), 20, 200 );

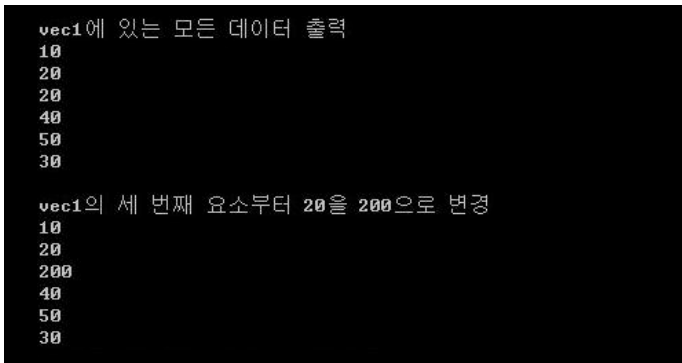
for( iterPos = vec1.begin(); iterPos != vec1.end(); ++iterPos )
{
    cout << *iterPos << " " << endl;
}

return 0;
}

```

---

[리스트 10-7]의 실행 결과는 다음과 같다.



```

vec1에 있는 모든 데이터 출력
10
20
20
40
50
30

vec1의 세 번째 요소부터 20을 200으로 변경
10
20
200
40
50
30

```

변경 가능 시퀀스 알고리즘에는 generate, copy, remove, replace 이외에도 fill, swap, reverse, transform 등이 있으니 여기서 설명하지 않은 알고리즘들은 다음에 여유가 있을 때 공부하기를 바란다.

지금까지 설명한 알고리즘을 보면 어떤 규칙이 있다는 것을 알 수 있다. 알고리즘에 넘기는 파라미터는 알고리즘을 적용할 컨테이너의 위치를 가리키는 반복자와 특정 값 또는 조건자를 파라미터로 넘기고 있다. 그래서 각 알고리즘은 이름과 기능이 다를 뿐 사용방법은 대체로 비슷하다. 즉 사용방법에 일괄성이 있다.

그래서 하나의 알고리즘만 사용할 줄 알게 되면 나머지 알고리즘은 쉽게 사용할 수 있게 된다. 이런 것이 바로 STL의 장점이라고 할 수 있다.

이번 장에서는 비슷한 성격의 알고리즘을 모아서 '변경 불가 시퀀스 알고리즘', '변경 가능 시퀀스 알고리즘', '정렬 관련 알고리즘', '범용 수치 알고리즘'으로 크게 네 개로 나누고 이 중 '변경 불가 시퀀스 알고리즘' 과 '변경 가능 시퀀스 알고리즘'에 있는 주요 알고리즘의 특징과 사용방법에 대해서 설명하였다.

이제 10.5장에서는 앞에서 설명하지 못한 '정렬 관련 알고리즘' 과 '범용 수치 알고리즘'의 주요 알고리즘의 특징과 사용방법에 대해서 설명하도록 한다.

## 10.5 정렬 관련 알고리즘

### 10.5.1 sort

sort는 컨테이너에 있는 데이터들을 내림차순 또는 오름차순으로 정렬 할 때 가장 자주 사용하는 알고리즘이다. 컨테이너에 저장하는 데이터의 자료 형이 기본 형이라면 STL에 있는 greater나 less 비교 조건자를 사용한다(STL의 string의 정렬에도 사용할 수 있다. 다만 이 때는 알파벳 순서로 정렬된다). 기본형이 아닌 경우에는 직접 비교 조건자를 만들어서 사용해야 한다.

---

sort의 원형

```
template<class RandomAccessIterator>
void sort( RandomAccessIterator _First, RandomAccessIterator _Last );
```

---

첫 번째와 두 번째 파라미터는 정렬하려는 구간의 시작과 마지막을 가리키는 반복자이다. 비교 조건자를 필요로 하지 않고 기본형을 저장한 컨테이너를 오름차순으로 정렬한다.

---

```
template<class RandomAccessIterator, class Pr>
void sort( RandomAccessIterator _First, RandomAccessIterator _Last,
BinaryPredicate _Comp );
```

---

첫 번째와 두 번째 파라미터는 정렬하려는 구간의 시작과 마지막을 가리키는

반복자이다. 세 번째 파라미터는 정렬 방법을 기술한 비교 조건자다.

sort 알고리즘의 원형을 보면 알 수 있듯이 랜덤 접근 반복자를 지원하는 컨테이너만 sort 알고리즘을 사용할 수 있다.

## sort 사용방법

---

```
vector<int> vec1;
...
// 오름차순 정렬
sort( vec1.begin(), vec1.end() );

// 내림차순 정렬
Sort( vec1.begin(), vec1.end(), greater<int>() );
```

---

[리스트 10-8] less와 greater 비교 조건자를 사용한 sort

---

```
#include <vector>
#include <iostream>
#include <algorithm>
#include <functional>

using namespace std;

int main()
{
    vector<int> vec1(10);
    vector<int> vec2(10);
    vector<int> vec3(10);
    vector<int>::iterator Iter1;

    generate( vec1.begin(), vec1.end(), rand );
    generate( vec2.begin(), vec2.end(), rand );
    generate( vec3.begin(), vec3.end(), rand );

    // 오름차순 정렬
    cout << "vec1 정렬 하기 전" << endl;
    for( Iter1 = vec1.begin(); Iter1 != vec1.end(); ++Iter1 ) {
```

---

---

```

    cout << *Iter1 << ", ";
}
cout << endl;

sort( vec1.begin(), vec1.end() );

cout << "vec1 오름차순 정렬" << endl;
for( Iter1 = vec1.begin(); Iter1 != vec1.end(); ++Iter1 ) {
    cout << *Iter1 << ", ";
}
cout << endl;
cout << endl;

// 내림차순 정렬
cout << "vec2 정렬 하기 전" << endl;
for( Iter1 = vec2.begin(); Iter1 != vec2.end(); ++Iter1 ) {
    cout << *Iter1 << ", ";
}
cout << endl;

sort( vec2.begin(), vec2.end(), greater<int>() );

cout << "vec2 내림차순 정렬" << endl;
for( Iter1 = vec2.begin(); Iter1 != vec2.end(); ++Iter1 ) {
    cout << *Iter1 << ", ";
}
cout << endl;
cout << endl;

// 일부만 내림차순 정렬
cout << "vec3 정렬 하기 전" << endl;
for( Iter1 = vec3.begin(); Iter1 != vec3.end(); ++Iter1 ) {
    cout << *Iter1 << ", ";
}
cout << endl;

sort( vec3.begin() + 5, vec3.end(), greater<int>() );

cout << "vec3 일부만 내림차순 정렬" << endl;

```

---



---

```

    for( Iter1 = vec3.begin(); Iter1 != vec3.end(); ++Iter1 ) {
        cout << *Iter1 << ", ";
    }
    cout << endl;

    return 0;
}

```

---

[리스트 10-8]의 실행 결과는 다음과 같다.

```

vec1 정렬 하기 전
41, 18467, 6334, 26500, 19169, 15724, 11478, 29358, 26962, 24464,
vec1 오름차순 정렬
41, 6334, 11478, 15724, 18467, 19169, 24464, 26500, 26962, 29358,

vec2 정렬 하기 전
5705, 28145, 23281, 16827, 9961, 491, 2995, 11942, 4827, 5436,
vec2 내림차순 정렬
28145, 23281, 16827, 11942, 9961, 5705, 5436, 4827, 2995, 491,

vec3 정렬 하기 전
32391, 14604, 3902, 153, 292, 12382, 17421, 18716, 19718, 19895,
vec3 일부만 내림차순 정렬
32391, 14604, 3902, 153, 292, 19895, 19718, 18716, 17421, 12382,

```

[리스트 10-8]은 기본 형 데이터를 정렬하는 예제이며, 이번에는 유저 정의 형 데이터를 정렬하는 예제를 확인해 보자.

[리스트 10-9] USER 구조체의 Money를 기준으로 정렬

---

```

#include <vector>
#include <iostream>
#include <algorithm>
#include <functional>

using namespace std;

struct USER
{
    int UID;

```

---

---

```

    int Level;
    int Money;
};

struct USER_MONEY_COMP
{
    bool operator()(const USER& user1, const USER& user2)
    {
        return user1.Money > user2.Money;
    }
};

int main()
{
    USER User1; User1.UID = 1;  User1.Money = 2000;
    USER User2; User2.UID = 2;  User2.Money = 2050;
    USER User3; User3.UID = 3;  User3.Money = 2200;
    USER User4; User4.UID = 4;  User4.Money = 1000;
    USER User5; User5.UID = 5;  User5.Money = 2030;

    vector<USER> Users;
    Users.push_back( User1 );  Users.push_back( User2 );
    Users.push_back( User3 );  Users.push_back( User4 );
    Users.push_back( User5 );

    vector<USER>::iterator Iter1;

    cout << "돈을 기준으로 정렬 하기 전" << endl;
    for( Iter1 = Users.begin(); Iter1 != Users.end(); ++Iter1 ) {
        cout << Iter1->UID << " : " << Iter1->Money << ", ";
    }
    cout << endl << endl;

    sort( Users.begin(), Users.end(), USER_MONEY_COMP() );

    cout << "돈을 기준으로 내림차순으로 정렬" << endl;
    for( Iter1 = Users.begin(); Iter1 != Users.end(); ++Iter1 ) {
        cout << Iter1->UID << " : " << Iter1->Money << ", ";
    }
}

```

---

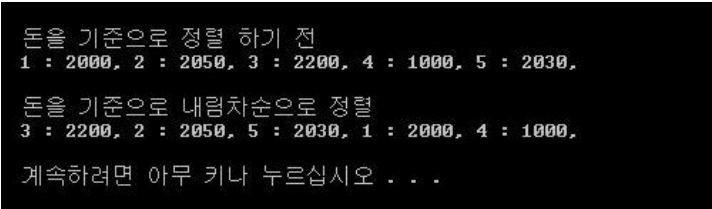
---

```
cout << endl << endl;

return 0;
}
```

---

[리스트 10-9]의 실행 결과는 다음과 같다.



```
돈을 기준으로 정렬 하기 전
1 : 2000, 2 : 2050, 3 : 2200, 4 : 1000, 5 : 2030,

돈을 기준으로 내림차순으로 정렬
3 : 2200, 2 : 2050, 5 : 2030, 1 : 2000, 4 : 1000,

계속하려면 아무 키나 누르십시오 . . .
```

## 10.5.2 binary\_search

이미 정렬 되어 있는 것 중에서 특정 데이터가 지정한 구간에 있는지 조사하는 알고리즘이다. 이것도 sort와 같이 비교 조건자가 필요 없는 버전과 필요한 버전 두 개가 있다(단 sort와 다르게 랜덤 접근 반복자가 없는 컨테이너도 사용할 수 있다).

---

```
binary_search 원형
template<class ForwardIterator, class Type>
bool binary_search( ForwardIterator _First, ForwardIterator _Last, const Type&
_Val );
```

---

첫 번째와 두 번째 파라미터는 조사하려는 구간의 시작과 마지막을 가리키는 반복자이다. 세 번째 파라미터는 조사할 값이다.

---

```
template<class ForwardIterator, class Type, class BinaryPredicate>
bool binary_search( ForwardIterator _First, ForwardIterator _Last, const Type&
_Val,
                    BinaryPredicate _Comp );
```

---

첫 번째와 두 번째 파라미터는 조사하려는 구간의 시작과 마지막을 가리키는 반

복자다. 세 번째 파라미터는 조사할 값을 가리키며, 네 번째 파라미터는 비교 조건자다.

---

binary\_search 사용방법

```
vector<int> vec1;
...
sort( vec1.begin(), vec1.end() );
...
bool bFind = binary_search( vec1.begin(), vec1.end(), 10 );
```

---

binary\_search는 정렬한 이후에 사용해야 한다고 앞서 이야기했다. 만약 정렬하지 않고 사용하면 어떻게 될까?

**[리스트 10-10]** 정렬하지 않고 binary\_search 사용

---

```
#include <vector>
#include <iostream>
#include <algorithm>
#include <functional>

using namespace std;

int main()
{
    vector<int> vec1;
    vec1.push_back(10);  vec1.push_back(20);  vec1.push_back(15);
    vec1.push_back(7);   vec1.push_back(100); vec1.push_back(40);
    vec1.push_back(11);  vec1.push_back(60);  vec1.push_back(140);

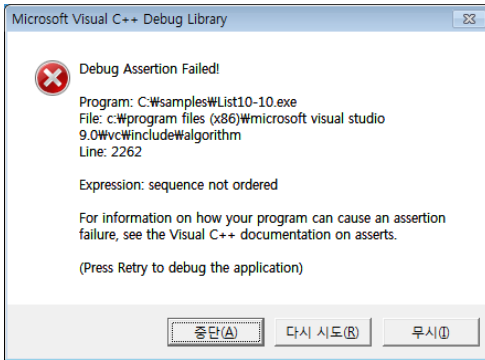
    bool bFind = binary_search( vec1.begin(), vec1.begin() + 5, 15 );
    if( false == bFind ) {
        cout << "15를 찾지 못했습니다." << endl;
    } else {
        cout << "15를 찾았습니다." << endl;
    }

    return 0;
}
```

---

디버그 모드로 빌드 후 실행하면 다음과 같은 에러 창이 나온다.

- 참고: Microsoft Visual C++ 2008 이상에서는 에러 없이 실행된다.



에러 내용은 시퀀스가 정렬되지 않았다고 나온다. 릴리즈 모드 빌드 후 실행하면 위와 같은 에러는 나오지 않지만 결과는 false가 나온다. `binary_search`를 사용할 때는 꼭 먼저 정렬해야 한다는 것을 잊지 말아야 한다.

[리스트 10-11] 정렬 후 `binary_search` 사용

---

```
#include <vector>
#include <iostream>
#include <algorithm>
#include <functional>

using namespace std;

int main()
{
    vector<int> vec1;
    vec1.push_back(10);  vec1.push_back(20);  vec1.push_back(15);
    vec1.push_back(7);   vec1.push_back(100); vec1.push_back(40);
    vec1.push_back(11);  vec1.push_back(60);  vec1.push_back(140);

    sort( vec1.begin(), vec1.end() );

    bool bFind = binary_search( vec1.begin(), vec1.begin() + 5, 15 );
```

---

---

```

if( false == bFind ) {
    cout << "15를 찾지 못했습니다." << endl;
} else {
    cout << "15를 찾았습니다." << endl;
}

return 0;
}

```

---

[리스트 10-11]의 실행 결과는 다음과 같다.

**15를 찾았습니다.**

비교 조건자를 사용하는 경우는 위의 [리스트 10-9] 코드를 예를 들면 [리스트 10-9]에서 사용했던 USER\_MONEY\_COMP 조건자를 사용한다.

[리스트 10-11], [리스트 10-9]의 Users를 binary\_search에 사용

---

```

.....
sort( Users.begin(), Users.end(), USER_MONY_COMP() );
bool bFind = binary_search( Users.begin(), Users.begin() + 3, User5,
USER_MONY_COMP() );
.....

```

---

## 10.5.3 merge

두 개의 정렬된 구간을 합칠 때 사용하는 것으로 두 구간과 겹치지 않은 곳에 합친 결과를 넣어야 한다. 주의해야 할 점은 합치기 전에 이미 정렬이 되어 있어야 하며 합친 결과를 넣는 것은 합치는 것들과 겹치면 안되며, 또한 합친 결과를 넣을 수 있는 공간을 확보하고 있어야 한다.

### merge의 원형

---

```

template<class InputIterator1, class InputIterator2, class OutputIterator>
OutputIterator merge( InputIterator1 _First1, InputIterator1 _Last1,
InputIterator2 _First2, InputIterator2 _Last2, OutputIterator _Result

```

---

---

```
);
```

---

첫 번째와 두 번째 파라미터는 하나의 입력 구간의 시작과 마지막을 가리키는 반복자이다. 세 번째와 네 번째 파라미터는 다른 입력의 시작과 마지막을 가리키는 반복자이다. 다섯 번째 파라미터는 합친 결과를 넣을 출력 구간의 시작을 가리키는 반복자다.

---

```
template<class InputIterator1, class InputIterator2, class OutputIterator, class
BinaryPredicate>
    OutputIterator merge( InputIterator1 _First1, InputIterator1 _Last1,
        InputIterator2 _First2, InputIterator2 _Last2, OutputIterator _Result,
        BinaryPredicate _Comp );
```

---

첫 번째와 두 번째 파라미터는 하나의 입력 구간의 시작과 마지막을 가리키는 반복자이다. 세 번째와 네 번째 파라미터는 다른 입력의 시작과 마지막을 가리키는 반복자이다. 다섯 번째 파라미터는 합친 결과를 넣을 출력 구간의 시작을 가리키는 반복자이다. 여섯 번째 파라미터는 비교 조건자다.

## merge 사용방법

---

```
vector<int> vec1, vec2, vec3;
...
merge( vec1.begin(), vec1.end(), vec2.begin(), vec2.end(), vec3.begin() );
```

---

다음 [리스트 10-12]는 vec1과 vec2를 합쳐서 vec3에 넣는 것으로 vec1과 vec2는 이미 정렬되어 있고 vec3는 이 vec1과 vec2의 크기만큼의 공간을 미리 확보해 놓고 있다.

[리스트 10-12] 두 개의 vector의 merge

---

```
#include <vector>
#include <iostream>
#include <algorithm>
#include <functional>

using namespace std;
```

---

---

```
int main()
{
    vector<int>::iterator Iter1;
    vector<int> vec1,vec2,vec3(12);

    for( int i = 0; i < 6; ++i )
        vec1.push_back( i );

    for( int i = 4; i < 10; ++i )
        vec2.push_back( i );

    cout << "vec1에 있는 값" << endl;
    for( Iter1 = vec1.begin(); Iter1 != vec1.end(); ++Iter1 ) {
        cout << *Iter1 << ", ";
    }
    cout << endl;

    cout << "vec2에 있는 값" << endl;
    for( Iter1 = vec2.begin(); Iter1 != vec2.end(); ++Iter1 ) {
        cout << *Iter1 << ", ";
    }
    cout << endl;

    merge( vec1.begin(), vec1.end(),
           vec2.begin(), vec2.end(),
           vec3.begin() );

    cout << "vec1과 vec2를 merge한 vec3에 있는 값" << endl;
    for( Iter1 = vec3.begin(); Iter1 != vec3.end(); ++Iter1 ) {
        cout << *Iter1 << ", ";
    }
    cout << endl;

    return 0;
}
```

---



[리스트 10-12]의 결과는 다음과 같다.

```
vec1에 있는 값
0, 1, 2, 3, 4, 5,
vec2에 있는 값
4, 5, 6, 7, 8, 9,
vec1과 vec2를 merge한 vec3에 있는 값
0, 1, 2, 3, 4, 4, 5, 5, 6, 7, 8, 9,
```

정렬 관련 알고리즘은 위에 소개한 세 개 이외에도 더 있지만 지면 관계상 보통 자주 사용하는 `sort`, `binary_search`, `merge` 세 개를 소개하는 것으로 마친다. 필자가 소개하지 않은 정렬 관련 알고리즘을 더 공부하고 싶은 분들은 MSDN이나 C++ 책을 통해서 공부하시기를 바란다.

## 10.6 범용 수치 알고리즘

### 10.6.1 accumulate

지정한 구간에 속한 값들을 모든 더한 값을 계산한다. 기본적으로 더하기 연산만 하지만 조건자를 사용하면 더하기 이외의 연산도 할 수 있다. `accumulate`를 사용하기 위해서는 앞서 소개한 알고리즘과 다르게 `<numeric>` 헤더 파일을 포함해야 한다.

#### accumulate의 원형

---

```
template<class InputIterator, class Type>
Type accumulate( InputIterator _First, InputIterator _Last, Type _Val );
```

---

첫 번째와 두 번째 파라미터는 구간이며, 세 번째 파라미터는 구간에 있는 값에 더할 값이다.

---

```
template<class InputIterator, class Type, class BinaryOperation>
Type accumulate( InputIterator _First, InputIterator _Last, Type _Val,
BinaryOperation _Binary_op );
```

---

네 번째 파라미터는 조건자를 사용하여 기본 자료 형 이외의 데이터를 더할 수 있고, 더하기 연산이 아닌 다른 연산을 할 수도 있다.

## accumulate 사용방법

---

```
vector<int> vec1;
...
// vec1에 있는 값들만 더한다.
int Result = accumulate( vec1.begin(), vec1.end(), 0, );
```

---

다음 [리스트 10-13]은 int를 저장하는 vector를 대상으로 accmurate를 사용하는 가장 일반적인 예다.

### [리스트 10-13] vector에 있는 값들을 계산

---

```
#include <vector>
#include <iostream>
#include <numeric>

using namespace std;

int main()
{
    vector<int>::iterator Iter1;
    vector<int> vec1;

    for( int i = 1; i < 5; ++i )
        vec1.push_back( i );

    // vec1에 있는 값
    for( Iter1 = vec1.begin(); Iter1 != vec1.end(); ++Iter1 ) {
        cout << *Iter1 << ", ";
    }
    cout << endl;

    // vec1에 있는 값들을 더한다.
    int Result1 = accumulate( vec1.begin(), vec1.end(), 0 );
```

---

---


```
// vec1에 있는 값들을 더한 후 10을 더한다.
int Result2 = accumulate( vec1.begin(), vec1.end(), 10 );

cout << Result1 << ", " << Result2 << endl;

return 0;
}
```

---

[리스트 10-13]의 실행 결과는 다음과 같다.



```
1, 2, 3, 4,
10, 20
```

이번에는 조건자를 사용하여 유저 정의형을 저장한 vector를 accumulate에서 사용해 보겠다. 이번에도 더하기 연산만을 했지만 조건자를 사용하면 곱하기 연산 등도 할 수 있다.

[리스트 10-14] 조건자를 사용한 accumulate

---

```
#include <vector>
#include <iostream>
#include <numeric>

using namespace std;

struct USER
{
    int UID;
    int Level;
    int Money;
};

struct USER_MONY_ADD
{
    USER operator()(const USER& user1, const USER& user2)
    {
```

---

---

```

    USER User;
    User.Money = user1.Money + user2.Money;
    return User;
}
};

int main()
{
    USER User1; User1.UID = 1; User1.Money = 2000;
    USER User2; User2.UID = 2; User2.Money = 2050;
    USER User3; User3.UID = 3; User3.Money = 2200;
    USER User4; User4.UID = 4; User4.Money = 1000;
    USER User5; User5.UID = 5; User5.Money = 2030;

    vector<USER> Users;
    Users.push_back( User1 ); Users.push_back( User2 );
    Users.push_back( User3 ); Users.push_back( User4 );
    Users.push_back( User5 );

    vector<USER>::iterator Iter1;

    for( Iter1 = Users.begin(); Iter1 != Users.end(); ++Iter1 ) {
        cout << Iter1->UID << " : " << Iter1->Money << ", ";
    }
    cout << endl << endl;

    // Users에 있는 Money 값만 더하기 위해 Money가 0인 InitUser를 세 번째 파라미터에,
    // 조건자를 네 번째 파라미터로 넘겼다.
    USER InitUser; InitUser.Money = 0;
    USER Result = accumulate( Users.begin(), Users.end(), InitUser,
    USER_MONY_ADD() );
    cout << Result.Money << endl;

    return 0;
}

```

---

[리스트 10-14]의 실행 결과는 다음과 같다.

```
1 : 2000, 2 : 2050, 3 : 2200, 4 : 1000, 5 : 2030,  
9280
```

## 10.6.2 inner\_product

두 입력 시퀀스의 내적을 계산하는 알고리즘으로 기본적으로는 +와 \*을 사용한다. 두 입력 시퀀스의 값은 위치의 값을 서로 곱한 값을 모두 더 한 것이 최종 계산 값이 된다. 주의 해야 할 것은 두 입력 시퀀스의 구간 중 두 번째 시퀀스는 첫 번째 시퀀스 구간 보다 크거나 같아야 한다. 즉 첫 번째 시퀀스 구간의 데이터는 5개가 있는데 두 번째 시퀀스에 있는 데이터가 5개 보다 작으면 안 된다.

### inner\_product의 원형

---

```
template<class InputIterator1, class InputIterator2, class Type>  
    Type inner_product( InputIterator1 _First1, InputIterator1 _Last1,  
        InputIterator2 _First2,  
        Type _Val );
```

---

조건자를 사용하는 버전으로 조건자를 사용하면 유저 정의형을 사용할 수 있는 내적 연산 방법을 바꿀 수 있다.

---

```
template<class InputIterator1, class InputIterator2, class Type,  
        class BinaryOperation1, class BinaryOperation2>  
    Type inner_product( InputIterator1 _First1, InputIterator1 _Last1,  
        InputIterator2 _First2,  
        Type _Val, BinaryOperation1 _Binary_op1, BinaryOperation2 _Binary_op2 );
```

---

다음 [리스트 10-15]은 조건자를 사용하지 않는 inner\_product를 사용하는 것으로 vec1과 vec2의 내적을 계산한다.

[리스트 10-15] inner\_product를 사용하여 내적 계산

---

```
#include <vector>
#include <iostream>
#include <numeric>

using namespace std;

int main()
{
    vector<int> vec1;
    for( int i = 1; i < 4; ++i )
        vec1.push_back(i);

    vector<int> vec2;
    for( int i = 1; i < 4; ++i )
        vec2.push_back(i);

    int Result = inner_product( vec1.begin(), vec1.end(), vec2.begin(), 0 );
    cout << Result << endl;

    return 0;
}
```

---

[리스트 10-15]의 실행 결과는 다음과 같다.

14

[리스트 10-14]의 vec1과 vec2에는 각각 1, 2, 3 의 값이 들어가 있다. 이것을 네 번째 파라미터의 추가 값을 0으로 넘긴 inner\_droduct로 계산하면 다음과 같다.

$$14 = 0 + (1 * 1) + (2 * 2) + (3 * 3);$$

[리스트 10-14]의 코드를 보기 전에는 어떻게 계산 되는지 잘 이해가 되지 않았다면 [리스트 10-14] 코드를 보았을 때 inner\_product가 어떻게 계산 되는지 쉽게 이해할 수 있게 된다.

inner\_product도 다른 알고리즘처럼 조건자를 사용할 수 있다. 필자가 앞서 다

른 알고리즘에서 조건자를 사용한 예를 보여 드렸으니 `inner_product`에서 조건자를 사용하는 방법은 숙제로 남겨 놓는다. 범용 수치 알고리즘에는 위에 설명한 `accumulate`와 `inner_product` 이외에도 더 있지만 다른 것들은 보통 사용 빈도가 높지 않고 그것들을 다 소개하기에는 많은 지면이 필요로 하므로 이것으로 끝을 맺는다.

범용 수치 알고리즘을 끝으로 STL의 알고리즘을 설명하는 것을 마친다. 이번 장에서 소개한 알고리즘은 STL에 있는 알고리즘 중 사용 빈도가 높은 알고리즘들로 이 것 이외에도 많은 알고리즘이 있으니 필자가 설명한 알고리즘을 공부한 후에 설명하지 않은 알고리즘들도 공부하시기를 바란다.

지금까지 설명한 글들을 보면 STL은 사용할 때 일관성이 높은 것을 알 수 있다. 높은 일관성 덕분에 하나를 알면 그 다음은 더 쉽게 알 수 있게 된다.

C++의 STL은 결코 사용하기 어려운 것이 아니다. C++을 알고 있다면 아주 쉽게 공부할 수 있으며 STL을 사용함으로 더 쉽고 견고하게 프로그래밍할 수 있다. 그러나 STL의 컨테이너나 알고리즘에 대해서 잘 모르면서 다른 사람들이 사용한 코드를 보고 그냥 사용하면 STL이 독이 될 수도 있음을 조심해야 한다.

필자는 몇 달 전부터 C++0x을 공부하고 있다. C++0x는 현재 개발중인 새로운 C++ 표준이다. C++0x에는 지금의 C++을 더 강력하고 쉽게 사용할 수 있게 해 주는 다양한 것들이 있다. 이 중 `lambda`라는 것이 있는데 이것을 사용하면 알고리즘에서 조건자를 사용할 때 지금보다 훨씬 더 편하게 기술할 수 있다. STL을 공부한 이 후에는 C++0x을 공부하시기를 바란다.

## 편집 후기

### 김제룡

대학생 시절에 잡은 평생 목표 중에 40살이 되기 전에 책을 쓴다는 내용이 있었습니다. 다른 목표들이 진행 중이기에 우선순위는 밀렸지만, 책을 만드는 과정이 궁금했습니다. 그런 와중에 인터넷에 좋은 글을 자주 쓰시는 최홍배님의 STL글 관련 EBOOK제작 내용을 접하게 되었습니다. 처음에는 STL책에 대한 리뷰정도로 생각했는데, 진행하다 보니 책을 만들기 위한 과정에서 신경 써야 할 것이나, 많은 분의 노력이 들어가게 된다는 것을 알게 되었습니다. 읽기 쉽게 잘 쓰인 글이어서 많은 수정을 할 필요도 없었고, 짧은 시간에 좋은 경험이 되었던 것 같습니다.

### 박민근

STL은 C++이 주무기인 게임 프로그래머에게 없어서는 안될 가장 중요한 라이브러리라고 해도 과언이 아닙니다. 그동안 STL을 기초부터 제대로 소개한 책이라고는 번역서뿐이었는데, 이렇게 직접 게임 서버 프로그래머인 최홍배님께서 작성한 내용이 책으로 나오게 되고 거기에 조그마한 도움이라도 드릴 수 있어서 좋았습니다.

### 이정재

STL은 C++ 표준 라이브러리에 속하는 부분으로 C++을 공부하기 위해서는 꼭 필요한 부분입니다. 그러나, STL에 대한 개념을 이해하고 적절한 곳에 잘 사용하기 위해서는 많은 연습이 필요한 라이브러리입니다. 이 번 작업을 통해 STL에 대한 이해를 넓힐 수 있는 좋은 경험이 되었습니다.



## 조진희

STL은 대학원 조교(자료구조, 알고리즘) 하면서 그리고 논문에 필요한 시뮬레이션 구현 시 사용해 왔습니다. 학부 때는 많이 쓰지 않았지만 연구, 개발 등 현업에 계신 분들은 자주 사용하실 거라 생각합니다. 제가 맡은 부분이 저는 잘 안 쓰는 set, algorithm 마지막 부분이어서 난감할 뻔 했지만^^; 전반적으로 사용 방법 등을 재점검 해보는 시간이 되었습니다. 독자에게도 많은 도움이 될 거라 생각합니다.




## 한상곤

책을 편집하는 입장과 책을 읽는 입장이 이렇게 다른지 몰랐습니다. 하지만 편집을 하면서 굉장히 재미있었습니다. 특히 소스코드를 하나하나 수정해 가면서 원작자의 의도를 해치지 않기 위해서 고심하면서 코드를 작성하면서 나름대로 템플릿에 대해서 많이 공부하게 되었습니다. 편집이라는 행위가 이렇게 섬세한 감성을 요구하는지 이번에 절실히 느낄 수 있었습니다.

# 독자와 함께 만듭니다

한빛미디어의 독자 서비스에 참여해 보세요

## \*한빛미디어 도서 구매 후 인증하세요!

-  **한빛이코인 적립!**
-  **부가서비스 이메일 자동 전송!**
-  **구매 도서 독자평가 실시!**

- 혜택1. 도서 구매 시 현금처럼 사용할 수 있는 한빛이코인을 적립해 드립니다.  
혜택2. 구매 도서에 대한 정오표, 부록 등 관련자료를 이메일로 자동 전송해 드립니다.  
혜택3. 구매 도서에 대한 독자평가를 실시, 기획에 적극 반영합니다.




인증방법: 한빛미디어 홈페이지 ▶ 로그인 ▶ 도서인증

[www.hanb.co.kr](http://www.hanb.co.kr)

## \*한빛리더스가 되세요!

### 한빛리더스(Readers&Leaders)란?

한빛미디어에서 출간된 도서에 대해 리뷰 등 객관적인 도서 사후 평가, 출판사의 기획과 마케팅 활동을 돕기 위한 미션을 수행하는 독자들의 모임입니다.  
한빛리더스는 독자들의 경험과 지식을 공유하여 더 좋은 도서를 출간하기 위한 독자들과의 커뮤니케이션 장입니다. 한빛미디어를 사랑하는 많은 독자들의 참여 바랍니다.

- |                                                                                                             |                                                                                                             |
|-------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------|
|  <b>신간도서 리뷰 및 오타자 등록</b> |  <b>설문조사 참여</b>          |
|  <b>한빛도서 토론회</b>         |  <b>미출간 도서 읽고 미션 수행</b>  |
|  <b>번개 미션 수행</b>         |  <b>기획의견 및 벤치마킹 아이디어</b> |



매년 2회 모집하며 한빛리더스는 페이스북을 통해 활동합니다.  
지금 홈페이지에서 모집 현황을 확인하세요!  
<http://facebook.com/hanbitreaders>

# 책으로 여는 IT 세상

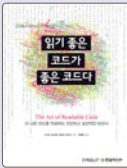
한빛미디어는 IT 개발자 여러분과 함께 발전해 왔습니다



## 프로그래머로 사는 법 : 프로그래머의 길을 걸어가는 당신을 위한 안내서

샘 라이트스톤 지음 | 서한수 옮김 | 25,000원 | **베스트셀러**

소프트웨어 업계에서 취직하는 방법, 높은 자리로 올라가는 방법, 최고 자리로 가는 방법처럼 프로그래머로 경력을 시작하는데 필요한 정보가 모두 담겨 있다. 스티브 워즈니악, 제임스 고슬링 등 세계적 프로그래머 구루의 독점 인터뷰가 담겨 있으며, 한국 개발자 9인의 인터뷰도 포함했다



## 읽기 좋은 코드가 좋은 코드다

: 더 나은 코드를 작성하는 간단하고 실전적인 테크닉

더스틴 보즈웰, 트레버 파우커 지음 | 임백준 옮김 | 18,000원 | **베스트셀러**

이 책은 코드를 작성할 때 언제나 적용할 수 있는 기본적인 원리와 실전적인 기술에 초점을 맞추고 있다. 누구나 쉽게 이해할 수 있는 코드를 예제로 사용하고, 각 장은 코딩과 관련한 다양한 측면을 파고든다. 그리하여 여러분이 어떻게 이해하기 쉬운 코드를 작성할 수 있는지를 보여준다.



## Make : Technology on Your Time Volume 04

메이크 미디어 지음 | 메이크 코리아 옮김 | 15,000원

3D 프린터를 중심으로, 탁상 제조 시대를 여는 톨과 기술을 '책상 위 공작소'라는 주제로 소개한다. 오라일리 미디어에서 2005년 창간한 MAKE 매거진은 한국에서는 한빛미디어에서 2011년 5월에 출간하였으며, 2012년에 국내 최초의 메이커페어인 '메이커페어 서울'을 개최했다.



## 린 스타트업 : 실리콘밸리를 뒤흔든 IT 창업 가이드

에시 모리아 지음 | 위선주 옮김 | 최환진 감수 | 18,000원

최고 벤처창업은 출발 당시 세웠던 플랜 A를 시의 적절하게 플랜 B, 플랜 C로 발전시키는 것을 '성공하는 비결'로 제시한다. 이 책은 플랜 A를 플랜 B, 플랜 C로 발전시키기 위한 사업 계획 수립과 고객 인터뷰 방법, 스타트업의 효율을 위한 업무 자침까지 저비용 고효율을 가능케 하는 린 스타트업의 진수를 보여준다.